НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

До захисту допущено:
Завідувач кафедри
_____ Олександр РОЛІК
«___»_____20__ р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Комп'ютеризовані системи управління»
спеціальності 151 «Автоматизація та комп'ютерно-інтегровані технології»
на тему: «Компілятор мови програмування загального призначення TinyLang на базі .NET»

Виконав:
студент IV курсу, групи ІА-62
Рибніков Владислав Ігорович _____

Керівник:
к.т.н., доцент
Креденцар Світлана Максимівна _____

Рецензент:
к.т.н., доцент
Лобанчикова Надія Миколаївна _____

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – перший (бакалаврський)
Спеціальність – 151 «Автоматизація та комп'ютерно-інтегровані технології»
Освітньо-професійна програма «Комп'ютеризовані системи управління»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олександр РОЛІК
«___»_____20__ р.

ЗАВДАННЯ

на дипломний проєкт студенту

Рибнікову Владиславу Ігоровичу

1. Тема проєкту «Контролер системи керування головкою протикорабельної ракети», керівник проєкту Креденцар Світлана Максимівна, к.т.н, доцент, затверджені наказом по університету від «_7_»_травня___ 2020 р. №1081-с_

2. Термін подання студентом проєкту ___09.06.2020____

3. Вихідні дані до проєкту Специфікація синтаксису мови програмування___

4. Зміст пояснювальної записки Вступ, огляд існуючих рішень та_____ формулювання задачі дипломного проєктування, основні принципи розробки компілятора, специфікація сформованої мови програмування, вирішення____ прикладних задач мовою TinyLang._____

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслеників, плакатів, презентацій тощо) діаграма взаємодії з інтегрованим середовищем розробки, схема алгоритму роботи компілятора, діаграма класів рівня синтаксичного аналізу, структурна схема роботи компілятора._____

6. Дата видачі завдання _____10.02.2020____

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Видача завдання дипломної роботи	10.02.2020	
2	Аналіз предметної області	2.03.2020	
3	Огляд існуючих мов програмування	5.03.2020	
4	Проєктування синтаксису	7.03.2020	
5	Проєктування компілятора	14.04.2020	
6	Розробка рівня лексичного аналізу	24.03.2020	
7	Розробка рівня синтаксичного аналізу	1.04. 2020	
8	Розробка рівня генерації коду	6.04.2020	
9	Проєктування та розробка середовища розробки	10.04.2020	
10	Тестування роботи мови програмування на прикладних задач	28.04.2020	
11	Опис роботи	10.05.2020	
12	Оформлення текстової та графічної документації в межах формування звіту	15.05.2020	

Студент Владислав РИБНІКОВ

Керівник Світлана КРЕДЕНЦАР

АННОТАЦІЯ

Рибніков В.І. Компілятор мови програмування загального призначення TinyLang на базі .NET. КПІ ім. Ігоря Сікорського, Київ, 2020.

Проєкт містить 76 с. тексту, 25 рисунків, 3 таблиці, посилання на 16 літературних джерела, 2 додатки та 4 конструкторських документа.

Ключові слова: компілятор, мова програмування, IDE, проміжний код, парсинг, синтаксичний аналіз, абстрактне синтаксичне дерево.

Об’єктом розробки є мова програмування TinyLang, а також компілятор до неї.

Мета розробки – полегшення вивчення мов програмування та пришвидшення розробки прикладних програм шляхом розробки синтаксису мови TinyLang та імплементації компілятора.

У дипломному проєкті розроблено ключові складові необхідні для написання коду, а саме: рівень, лексичного аналізу вхідного тексту, рівень синтаксичного аналізу та рівень парсингу, а також був створений синтаксис мови програмування та інтегроване середовище розробки. Проведено аналіз існуючих мов програмування а також компіляторів до них та виділено ключові тенденції їх розвитку. Значну увагу було приділено декларативності та простоті синтаксису. Також більшість синтаксичних конструкцій було винесено на етап компіляції коду, що значну пришвидшує його написання.

Отримані результати можуть бути корисними при створенні аналогічних компіляторів, а також доповненні вже існуючих.

SUMMURY

Rybnikov V.I. TinyLang general purpose programming language compiler based on .NET. Igor Sikorsky KPI, Kyiv, 2020.

The project contains 76 p. text, 25 figures, 4 tables, references to 16 literature sources, 2 appendices and 4 design documents.

Keywords: compiler, programming language, IDE, intermediate code, parsing, parsing, abstract syntax tree.

The object of development is the TinyLang programming language and its compiler.

The purpose of the development is to facilitate the study of programming languages and to accelerate the development of application programs by developing the syntax of the TinyLang language and implementing the compiler.

The diploma project developed the key components necessary for writing code, namely: level of lexical analysis of the input text, level of parsing and intermediate code generating level, as well as the syntax of the programming language and the integrated development environment. The analysis of existing programming languages and also compilers to them is carried out and the key tendencies of their development are allocated. Considerable attention was paid to declarativeness and simplicity of syntax. Also, most of the syntactic constructions were made at the stage of compiling code, which significantly speeds up its writing.

The results obtained can be useful in creating analogical compilers, as well as supplementing existing ones.

Номер рядка	Формат	Позначення	Найменування	Кільк. аркушів	Номер екзем.	Приміт-ка
1			Документація загальна			
2						
3			Знову розроблена			
4						
5	А4	ІА62.220БАК.005 ПЗ	Пояснювальна записка	76		
6	А3	ІА62.220БАК.005 Д1	Діаграма взаємодії з інтегро-	1		
7	А3		середовищем розробки			
8	А3	ІА62.220БАК.005 Д2	Діаграма алгоритму	1		
9			роботи компілятора			
10	А3	ІА62.220БАК.005 Д3	Діаграма класів рівня синтак-	1		
11			синтаксичного аналізу			
12		ІА62.220БАК.005 Д4	Діаграма компонентів	1		
13			компілятора			
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						
28						

Зм.	Аркуш	№ докум.	Підпис	Дата	ІА62.220БАК.005 ТП		
Розроб.		Рибніков В. І.			Компілятор мови програмування загального призначення TinyLang на базі .NET Відомість технічного проєкту	Літ.	Аркуш	Аркушів
Перевір.		Креденцар С. М.				Т		1
Реценз.						КПІ ім. Ігоря Сікорського, ФІОТ Група ІА-62		
Н. Контр.								
Затв.								

Пояснювальна записка

до дипломного проєкту

на тему: «Компілятор мови програмування загального призначення TinyLang на базі .NET»

Київ – 2020 рік

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

AST – Abstract Syntax Tree

GL – Generation Level

IL – Intermediate Language

CIL – Common Intermediate Language

OS – Operation System

IoT – Internet of Things

PaaS – Platform as a Service

SaaS – Software as a Service

DSL – Domain Specific Language

EDSL – Embedded Domain Specific Language

ADT – Algebraic Data Types

SQL – Structured Query Language

JVM – Java Virtual Machine

LINQ – Language Integrated Query

JSON – JavaScript Object Notation

IDE – Integrated Developer Environment

ООП – Об’єктно-Орієнтоване програмування

ФП – Функціональне програмування

ВСТУП

Однією з основних сильних сторін сучасних електронних цифрових пристроїв є їх здатність програмуватися на виконання різноманітних корисних і розрізнених функцій.

Спочатку розроблені як «супер-калькулятори» для обмеженого використання у військових та наукових обчисленнях, комп'ютери стали однією з найбільш поширених технологій сучасного суспільства. Що робить комп’ютер таким потужним, це його величезна гнучкість: за умови відповідного програмного забезпечення недорогий і масово виготовлений комп'ютерний чіп може імітувати функцію багатьох більш дорогих пристроїв спеціального призначення.

Оскільки комерційні комп’ютери та контролери ставали менш дорогими і широко використовувались корпораціями, зацікавленими в комп’ютеризованій обробці даних та автоматизації виробничих процесів, потреба в нових методах програмування ставала все більш очевидною.

Виробники комп'ютерів хотіли забезпечити доступність їх пристроїв для максимально широкого кола споживачів тому перед програмістами поставала задача усунути складності, пов'язані з написанням та відлагодженням машинного коду. З’явилася ціла низка нових продуктів, спрямованих на те, щоб зробити програмування менш важким та трудомістким. По мірі того, як проєкти програмування стали більшими та складнішими, витрати на розробку програмного забезпечення різко зросли; до середини 1950-х років було підраховано, що програмування та налагодження становлять приблизно три чверті вартості роботи з комп'ютером.

Для вирішення проблеми складності написання програм були створення різного роду транслятори з більш високорівневих мов, які були схожі на людську мову в машинний код.

Розвиток високорівневих мов програмування дуже сильно вплинув на прийняття комп’ютерних технологій сучасним суспільством. На початку 50-х

років багатьом людям було зовсім не зрозуміло, для чого потрібні комп'ютери або як їх найкраще використовувати. Стандартизовані мови програмування, такі як Fortran та Cobol, дозволили цим компаніям ділитися своїм програмним забезпеченням та досвідом з іншими, тим самим заохочуючи поширення інформації, персоналу та методик. Кожна з багатьох мов програмування, розроблених у цей період, слугувала різному призначенню: Fortran дав змогу вченим ефективніше використовувати комп’ютери; Cobol надав функції, властиві обчислювальним потребам бізнесу; Basic та Pascal дозволили університетам підготувати наступне покоління програмістів. Відхід від машинного коду до більш алгебраїчних та нативних для людини висловлювань розширив базу користувачів комп'ютерів та зробив комп'ютери більш доступними та зрозумілими широкій аудиторії. Нарешті, накопичення досвіду та методик програмування дозволило забезпечити подальший розвиток більш досконалого програмного забезпечення.

З розвитком програмування сформувалися різноманітні підходи до розробки. Серед різноманітних трансляторів машинного коду виділилися: ассемблери, інтерпретатори та компілятори; серед підходів до розробки виокремились такі підходи як: функціональне програмування, об’єктно-орієнтоване, процедурне та інші. Все це ускладнює розробки програмного забезпечення та відходить від концепту стандартизації, який сформувався на етапі зародження комп’ютерної техніки. Багато існуючих мов програмування все ж таки являються мультипарадигмними та можуть застосовуватися для найрізноманітніших задач, але зазвичай поріг входження в такі мови залишається досить високим. З іншої сторони з достатнім успіхом існують вузько-спеціалізовані мови які мають досить обширний функціонал та не великий для вивчення обсяг синтаксису, але все рівно залишаються зрозумілими лише спеціалістами в одній конкретній області. В підсумку для середньостатистичного офісного користувача комп’ютера, або ж працівника на виробництві де використовуються комп’ютеризовані технології, світ програмування залишається досить складним.

Постає задача розробки мови програмування з максимально простим синтаксисом але з використанням всіх переваг сучасних компіляторів, що дозволяло б описувати вирішення складних бізнес проблем або ж написання алгоритмів для роботи автоматизованої техніки в декілька рядків.

Для досягнення запропонованої цілі потрібно сформувати досить зрозумілий та в той же час потужний синтаксис, а також написати компілятор для нього, який для підвищення гнучкості та підтримуваності буде працювати на вже існуючій платформі.

Для забезпечення зрозумілості синтаксису були взяті знайомі більшості C-подібні конструкції такі як: цикли for/while, оператори розгалуження if/elif/else та ін. Також були додані більш нові конструкції з функціональних мов програмування такі як: лямбда вирази, типи-записи, алгебраїчні типи.

Для написання компілятору була вибрана мова C#, яка дозволила більш декларативно описувати особливості синтаксису та алгоритми парсингу та трансляції. Також ключовим момент являється використання підходу парсер-комбінаторів, що зменшило складність програмування алгоритмів парсингу.

Розроблений після етапу парсингу, проводить певні оптимізації та спрощення різних конструкцій та транслює побудоване абстрактне синтаксичне дерево в мову низького рівня IL, що забезпечую повну сумісність сформованої мови програмування з уже існуючими на платформі .Net.

Бакалаврський проєкт складається з наступних розділів: вступ, основні розділи, висновки, список використаних джерел із 16 найменувань, 2 додатків. графічна частина включає 4 кресленика формату А. Загальний обсяг 106 сторінок

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ФОРМУЛЮВАННЯ ЗАДАЧ ДИПЛОМНОГО ПРОЄКТУВАННЯ

1.1 Історія розвитку програмування

Вже в першій половині XIX століття людство задумувалося над автоматизацією складних обрахунків. Саме тоді було створено першу механічну машину для обрахунків, а також першу на думку більшості мову програмування, яка була створена в 1883 році, коли жінка на ім'я Ада Лавлейс працювала з Чарльзом Беббіджем на його дуже ранній механічній обчислювальній аналітичній машинні. У той час як Беббідж займався простим обчисленням чисел, Лавлейс побачила, що цифри, з якими працював комп'ютер, можуть виражати дещо більше, ніж просто кількість. Вона написала алгоритм для аналітичної машини, який був першим у своєму роді. Через свій внесок Лавлейс приписують створення першої мови комп'ютерного програмування, саме вона вперше ввела такі терміни як “цикл”, “робоча комірка та описала основні принципи алгоритмізації.

Наступним великим кроком стала розробка мов ассемблера в 1950-х та написання перших трансляторів для них, які генерували машинний (двійковий) по заданим командам. До цього програмістам потрібно було запам’ятувати складні коди операцій. Мови ассемблера вважаються низькорівневими, так як вони хоч і являються зрозумілими людині, все ж досить схожі на машинний код так як це полегшує трансляцію. Розробка мов ассемблера дуже сильно вплинула на подальший розвиток низько-рівневих мов програмування

Після цього, починаючи з 60-х років, активно почали з’являтися більш поширені та широко вживані на той час мови такі як: Fortran, Algol, Lisp, Cobol, SmallTalk, C, SQL. Починали формуватися різноманітні підходи до розробки, а також створювалися мов, які вирішували лише конкретні задачі. Всі ці мови вже менше нагадували машинний код та мови ассемблеру, команди нагадували звичну людську мову. Все це в комплексі сприяло

розвитку програмування в цілому та збільшувало інтерес до цієї галузі.

В 90-х роках розвиток мов програмування досяг свого піку. Були сформовані основні парадигми програмування: об’єктно-орієнтоване, функціональне та процедурне програмування. З’явилася величезна кількість патернів, більшість бізнес процесів та виробництв було автоматизовано за допомогою комп’ютеризованої техніки та програмування. Також з поширення інтернету формувалася величезна галузь інтернет програмування та створювалися мови які могли працювати в браузері та сформували сучасне уявлення про глобальну мережу.

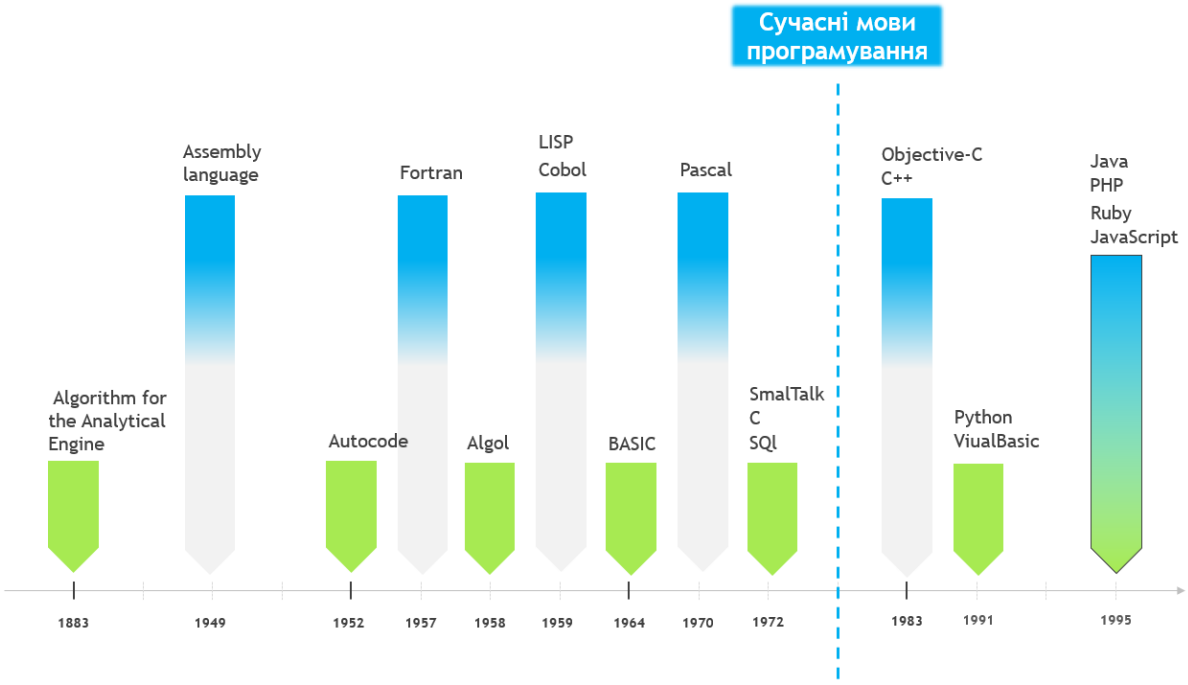

Рисунок 1.1 – Історія формування мов програмування

1.2 Основні види мови програмування

Дивлячись на розвиток мов програмування, умовно можна виділити певні етапи їх формування. Кожен етап значно вдосконалював як синтаксис так і можливості мов. Такі етапи формування прийнято назвати поколінням. Кожне з поколінь мови програмування має на меті забезпечити більш високий рівень абстрагування внутрішніх деталей комп'ютерного обладнання, роблячи

мову більш зручною для програмістів, потужною та універсальною. Сьогодні можна виділити п'ять основних поколінь які формують 2 основні відмінні між собою групи:

а) низькорівневі мови програмування:
   1) перше покоління (1GL) – машинний код;
   2) друге покоління (2GL) – ассемблер;

б) нисокорівневі мови програмування:
   1) третє покоління (3GL) – машинно-незалежні мови;
   2) четверте покоління (4GL) – доменно-специфичні мови;
   3) п’яте покоління (5GL) – логічне програмування.

Мови (програмування) першого покоління (1GL) - це група мов програмування, які є мовами машинного рівня, що використовуються для програмування комп'ютерів першого покоління. Мови цього рівня характерні відсутністю жодних трансляторів чи компіляторів при виконанні. Інструкції програм першого покоління зазвичай вводилися через перемикачі на передній панелі комп'ютерної системи.

Інструкції в 1GL складаються з двійкових чисел, представлених як 1 та 0. Це робить мову придатною для розуміння машиною, але набагато складніше інтерпретується та вивчається людиною.

Мови програмування другого покоління також належить до категорії мов програмування низького рівня. Мови другого покоління включають мови ассемблеру, які використовують концепцію мнемоніки для опису програм. В таких мовах символічні назви використовуються для представлення опкоду та операндної частини інструкції, яка потім транслюється в машинні інструкції.

Покоління 2GL вже має значні переваги над поколінням першого рівня і забезпечує:

— більш швидку розробку та підтримуваність з боку програмістів;
— краще виявлення помилок в алгоритмах програми;
— знижену складність вивчення;
— друге покоління досить широко використовується і досі при

— програмуванні мікроконтролерів та мікропроцесорів.

Мови третього покоління 3GL є набагато більш незалежними від машин та більш зручними для програмістів. З’являються такі можливості, як покращена підтримка агрегованих типів даних та вираження понять таким чином, що зміщує орієнтованість програми з комп'ютера на розробника. Мови третього покоління суттєво вдосконалюється в порівнянні з мовами другого покоління завдяки тому, що всі несуттєві деталі програми такі як адресація комірок пам’яті та переміщення по стеку викликів виконується комп’ютером. 3GL є більш абстрактними, ніж мови попередніх поколінь, і тому мови цього покоління вже можна вважати мовами вищого рівня. Вперше представлені наприкінці 50-х років, Fortran, ALGOL та COBOL - приклади раннього 3GL.

Найбільш популярні сьогодні мови загального призначення, такі як C, C++, C#, Java, BASIC та Pascal, також є мовами третього покоління, хоча кожну з цих мов можна додатково розподілити на інші категорії на основі різних спільних ознак. Більшість 3GL підтримують структурне програмування. Багато підтримують об'єктно-орієнтоване програмування. Такі ознаки частіше використовуються для опису конкретної мови, а не 3GL в загальному. Такий поділ також називається поділом по парадигмам програмування. Виділяють 2 основні групи мов програмування: процедурне та декларативне, які досить сильно відрізняються один від одного на концептуальному рівні. При процедурному програмуванні код складається з послідовних викликів певних блоків (операцій), які трохи нагадують попередній рівень мов ассемблеру.

Декларативне ж програмування оперує в основному синтаксичними конструкціями які формують код досить схожий на набір речень людської мови. Тобто при процедурному програмуванні програміст задає конкретні операції, які має виконати комп’ютер, а при декларативному описує логіку алгоритму без використання деталей у вигляді конструкцій: if/else, for, while та ін. Зазвичай, декларативно написана програма виглядає як речення звичайною, природною мовою, прочитавши яке, можна швидко зрозуміти

роботу програми.

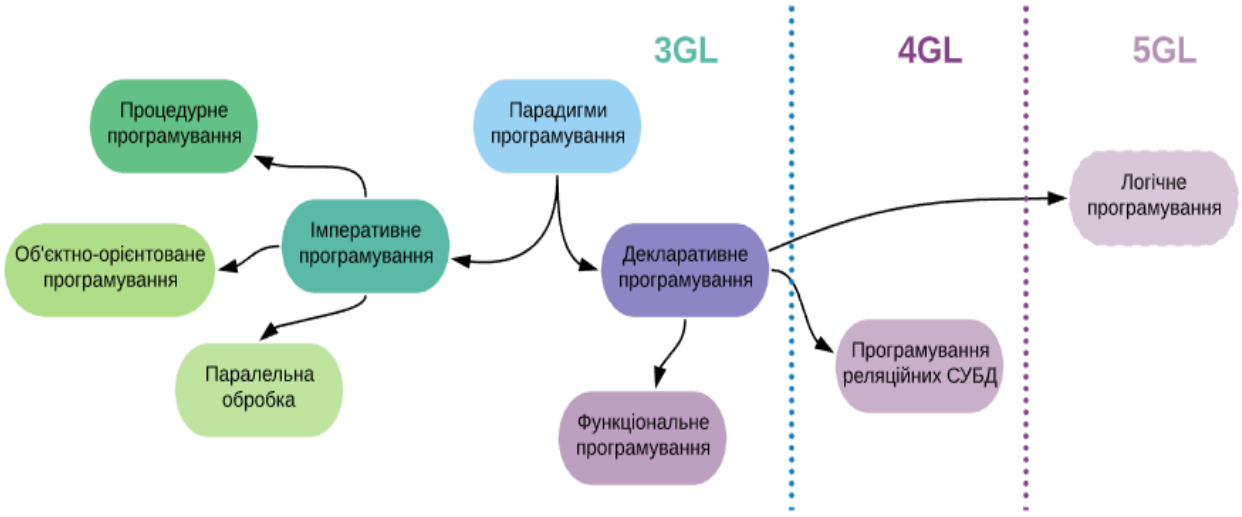

Рисунок 1.2 – Поділ мов програмування за основними парадигмами

Як бачимо поділ за парадигмами не існує лише в рамках одного 3 рівня та перетинається з іншими. В більшості своїй, декларативні мови відносяться до більш високих рівнів 4GL та 5GL.

Мова програмування четвертого покоління (4GL) - це будь-яка мова комп'ютерного програмування, що належить до класу мов, що передбачається як певне вдосконаленням мови програмування третього покоління (3GL) для специфічного домену. Мови 4GL можуть включати підтримку управління базами даних, створення звітів, математичну оптимізацію, розробка графічного інтерфейсу та загалом мають назву DSL.

Сьогодні сюди можна також віднести підмножини синтаксису або специфічні бібліотеки в рамках однієї мови програмування, або так звані EDSL (Embedded Domain Specific Language). Прикладом може бути синтаксис LINQ в мові програмування C#, який додає підтримку SQL-подібних запитів до даних: як в середині програми (LINQ to IEnumerable) так і до зовнішніх даних (LINQ to IQueryable). Останній в свою чергу використовує досить цікавий та потужний підхід до трансляції таких запитів в синтаксис мови вихідного запиту (наприклад TSQL у випадку з базами даних MS SQL) за допомогою спеціальних провайдерів, підміняючи які можна транслювати LINQ запит практично в будь-яку іншу мову.

Мова програмування п'ятого покоління (5GL) - це будь-яка мова програмування, заснована на вирішенні проблем, використовуючи обмеження, задані програмою, а не використовуючи алгоритм, написаний програмістом. Більшість мов програмування, заснованих на обмеженнях, та деякі інші декларативні мови - це мови п'ятого покоління.

Хоча мови програмування четвертого покоління призначені для побудови конкретних програм, мови п'ятого покоління призначені для того, щоб комп'ютер вирішив задану проблему без програміста. Таким чином, користувачеві потрібно лише турбуватися про те, які проблеми потрібно вирішити та які умови потрібно виконувати, не турбуючись про те, як реалізувати розпорядок чи алгоритм їх вирішення. Мови п’ятого покоління в основному використовуються в дослідженнях штучного інтелекту.

1.3 Визначення та історія розвитку компіляторів

У загальному вигляді компілятор - це програма, яка приймає як вхід текст іншої програми, описаної мовою певного рівня абстракції та перетворює його в програмний код більш низького рівня, зберігаючи значення цього тексту. Цей процес також називається трансляцією або більш звично нам - перекладом, як це було б, якби тексти були природними мовами. Практично всі компілятори перекладаються з однієї мови введення, мови-джерела в єдину мову виводу , а мова на якій написаний сам компілятор, є мовою реалізації.

Процес компіляції можливий завдяки наступним факторам:

— мова джерело та семантика вводу є визначеною та описується певною специфікацією;

— цільова мова також є чітко визначеною.

Компілятор складається із серії модулів, які перетворюють, вдосконалюють та передають інформацію між собою. Інформація передається переважно послідовно, від модуля Mn до модуля Mn + 1. Кожна така пара послідовних модулів визначає інтерфейс та структуру компілятора, а інформація

яка передається між ними являється завершеною формою даних певного типу [2].

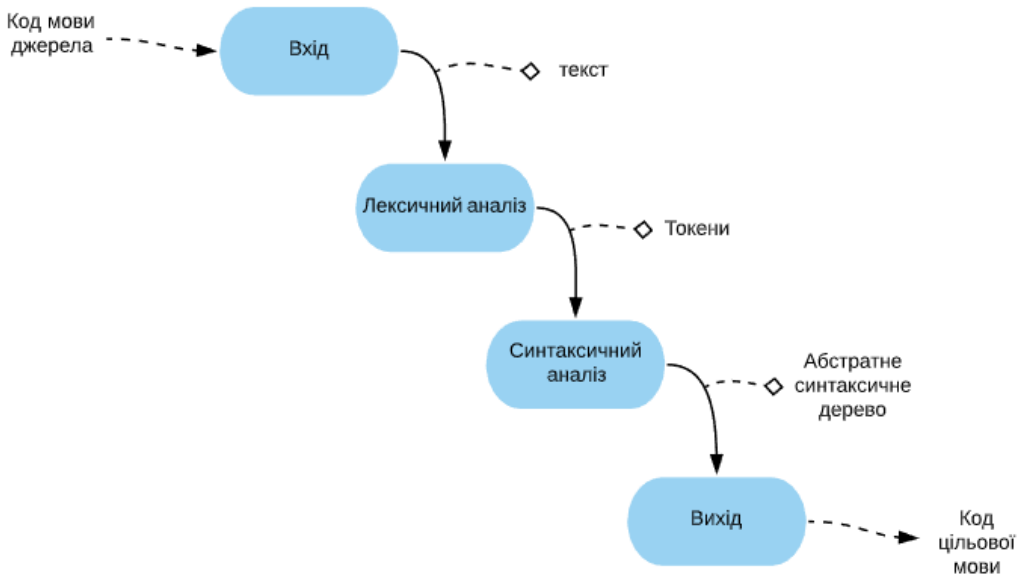

Рисунок 1.3 – Спрощена схема роботи компілятора

Модуль введення програмного тексту - знаходить текст програми, читає його ефективно і перетворює його в потік символів, дозволяючи отримувати набір рядків. Він також може переходити на інші файли, коли їх потрібно включити. Цей модуль працює з операційною системою з одного боку та з лексичним аналізатором з іншого.

Модуль лексичного аналізу виділяє лексеми у вхідному потоці та визначає їх клас та представлення. Він може бути написаний від руки або згенерований з опису лексем. Крім того, він може зробити деяку обмежену інтерпретацію на деяких маркерах, наприклад, щоб побачити, чи є ідентифікатор ідентифікатором макросу або ключовим словом (зарезервоване слово).

Модуль аналізу синтаксису структурує потік лексем у відповідне абстрактне дерево синтаксису (AST). Деякі аналізатори синтаксису складаються з двох модулів. Перший зчитує потік токенів і викликає функцію з другого модуля для кожної конструкції синтаксису, яку він розпізнає; функції другого модуля потім будують вузли AST і зв'язують їх.

Цей опис роботи компілятора на сьогодні є досить поверхневим і не розкриває всі задачі та можливості сучасних компіляторів, але показує основний принцип послідовності перетворення інформації одного виду в інший.

На сьогодні можна виділити наступні тенденції в дизайні компіляторів:

— акцент на оптимізації згенерованого коду;

— спрощення синтаксису;

— використання проміжного коду;

— використання високорівневої мови реалізації;

— написання компілятору використовуючи мову реалізацію таку ж як мову джерело.

1.4 Огляд існуючих мов програмування, визначення тенденцій їх розвитку та основних недоліків

В останні роки акцент програмування все більше зміщується в сторону обробки паралельної обробки великих масивів даних, а також з’являються зовсім нові види задач та типи програм. Набирають досить велику популярність хмарні обчислення та використання так званих “хмарних функції” – невеликі за обсягом програми, які використовують хмарні технології та повністю абстраговані від платформи на якій вони виконуються. Популяризуються задачі IoT та програмування роботизованих засобів автоматизації.

Щоб залишатися актуальними у новому просторі програмування, було докладено чимало зусиль для оновлення старих мов, таких як Java та C ++, щоб відповідати поточному стану справ. Наприклад, у Java, у версії Java 8 додано Lambda Expressions та Streams API, а Java 9 тепер поставляється із REPL. Мультипотокові бібліотеки також були включені до мов для підвищення їх одночасності. Основним недоліком цих «старих» мов є те, що вони ніколи не будувалися з нуля для вирішення виникаючих проблем у сучасному світі програмування. Спроба вдосконалити нові функції на мовах також не виявилася

елегантним рішенням. У нас не залишається іншого вибору, крім використання нових мов програмування, які були розроблені з самого початку, щоб вирішити деякі важкі проблеми сучасної інженерії програмного забезпечення.

Все ці зміни сильно ускладнюють існуючі мови та роздувають їх досить складними конструкції. Все це досить очевидно ставить задачу формування нових мов.

Зважаючи на це, було винайдено безліч нових мов програмування для того, щоб спробувати вирішити існуючі проблеми. Ці мови можна умовно виділити в категорію «сучасних», оскільки всі вони були випущені протягом цього століття. Більшість цих нових мов мають багато спільного. Синтаксис деяких з них виглядає дуже схоже. Деякі спільні риси, якими користується більшість мов, є:

— змінні повинні бути незмінними, або ж іммутабельними (англ - immutable) по замовчуванню

— автоматичний вивід типів;

— наголошення на безпеці типів;

— деякі з них мають спрощене керування потоками;

— більшість з них наголошує на функціональному стилі програмування;

— більшість з них не вимагають крапки з комою як термінатору тверджень;

— більшість з них мають REPL (англ. Read-eval-print loop - цикл читання-обчислення-друку);

— більшість з них статично типізовані;

— більшість з них мають чистий і вишуканий синтаксис, без особливого безладу і багатослівності [16].

Попри всі перелічені особливості, сучасні мови програмування мають ряд спільних недоліків, які створюють потребу в інакшому підході в дизайні мови програмування:

а) стандартизація – в сучасному програмуванні існує безліч мов, які вирішують конкретні задачі, але вони суттєво відрізняються між собою. Особливо помітна різниця в скриптових мов для автоматизації, а також системних та мовах для розробки застосунків;

б) інтегрованість – підхід до компіляції та виконання програм теж сильно відрізняється, тому наприклад ми не можемо використовувати бібліотеку написану мовою Python в програмах написаних на JavaScript;

в) складність синтаксису – поріг входження до вивчення нових мов програмування сьогодні досить високий та потребує базисних знань платформи та синтаксису конкретної мови, а також загальні знання з інформатики.

Розробка нової мови програмування має спростити та стандартизувати розробку. Одним з найбільших її переваг буде об’єднання 3-х підходів: скриптового, об’єктно-орієнтованого, функціонального. Це дозволить додати можливостей при розробці скриптів автоматизації та водночас спростить розробку повномасштабних застосунків. Акцентом має стати декларативність, проблемно-орієнтованість та потужність синтаксису, що дозволить віднести мову програмування до класу 4GL, який відзначається зрозумілістю синтаксису та можливостями компіляції.

1.5 Постановка задач дипломного проєктування

Розглядаючи всі проблеми, а також виділивши основні акценти, постає задача проєктування нової, сучасної мови програмування, написаної з використанням сучасних можливостей високорівневих мов та зі спрощеним синтаксисом, що підвищить декларативність та зрозумілість мови.

Задачі дипломного проєктування:

а) розробка синтаксису нової мови програмування TinyLang, яка поєднує у собі скриптовий, об’єктно-орієнтований та функціональний підходи;

б) розробка компілятору мови програмування TinyLang з використанням комбінаторів парсерів;

в) створення інтегрованого середовища розробки, яке надає функціонал компіляції та виконання програм мовою TinyLang;

г) апробація розробленої мови програмування для вирішення прикладних задач.

В цілому сама назва мови буде натякати на акценті спрощеного та зрозумілого синтаксису (TinyLang – англ: “Крихітна мова”). Спрощення старих конструкцій та додавання нових дозволить мові виглядати “свіжою”, але без зайвих синтаксичних ускладнень та з підтримкою усіх можливостей сучасних мов програмування. Існування даної мови в рамках платформи .NET дозволить пришвидшити її популяризацію серед вже сформованої спільноти розробників та в майбутньому можливість постачатись в рамках розширення до середовища розробки Visual Studio.

Немало важливим буде застосування комбінаторів парсингу та мови C# при написанні, що значно пришвидшує саму розробку а також додає можливість включити компілятор як бібліотеку в будь-яку іншу .NET-сумісну мови. Це відкриває можливості до розвитку мови не тільки самостійно, а й в якості EDSL розширення до мови C#. Також відкритий API компілятору, схожий на аналогічний в Roslyn, та постачання коду компілятору в форматі Open Source на платформі GitHub дозволить іншим розробникам легко додавати нові можливості до мови та покращувати швидкодію компілятора.

Спрощений синтаксис, потужність компілятору, відкритість платформи та відносно новий підхід до розробки який поєднує можливості скриптових, об’єктно-орієнтованих та функціональних мов дозволить швидко підвищити популярність мови та компілятору, що сприятиме розширенню функціоналу та розвитку синтаксису.

2 ОСНОВНІ ПРИНЦИПИ РОЗРОБКИ КОМПІЛЯТОРА

Основою розробки будь-якої мови програмування є її компілятор, який дозволяє перетворювати її в менш високорівневу мову яка може бути виконана за допомогою певного програмного забезпечення.

Першочерговою властивістю хорошого компілятора є звичайно те, що він генерує правильний код. Компілятор, який час від часу генерує неправильний код, марний; компілятор, який генерує неправильний код раз на рік, може здатися корисним, але небезпечним. Тобто будь який компілятор можна вважати чистою функцією, тобто для нього будуть виконуватись 2 умови:

— детермінованість;

— відсутність побічних ефектів.

Важливо також, щоб компілятор повністю відповідав мовній специфікації. Тобто для будь-якого виразу, написаного згідно зі специфікацією мови джерела, має бути згенерована відповідна конструкція цільової мови.

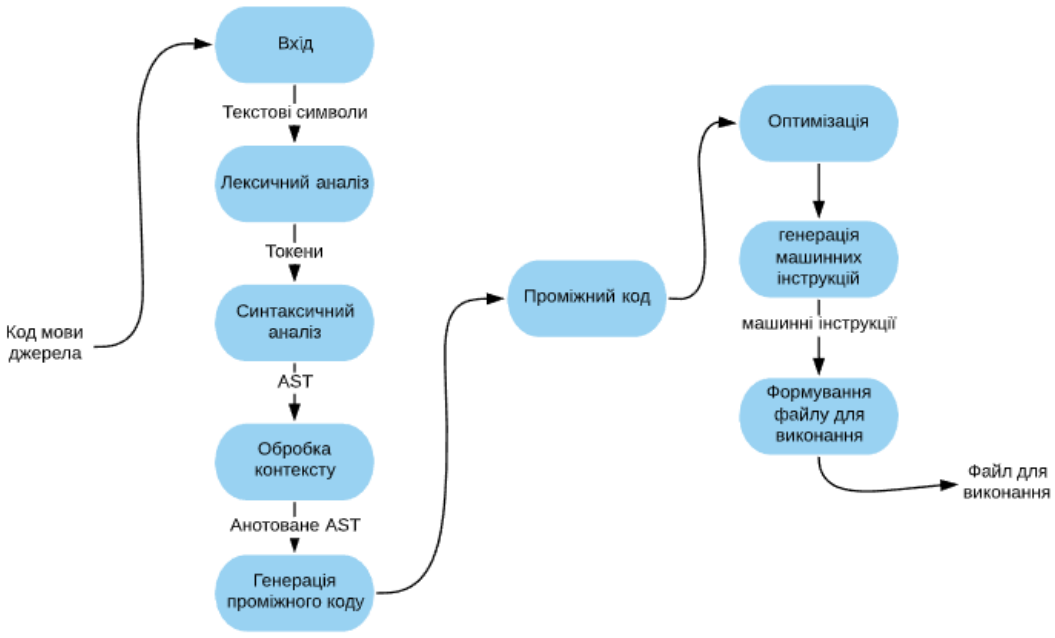

Рисунок 2.1 – Схема роботи компілятора від мови джерела до виконуваної програми

Приведена схема являється розширеним варіантом зазначеної в першому розділі (рисунок 2.1) та показує роль проміжного коду в процесі компіляції від текстового файлу до виконуваного.

2.1 Опис граматики для мови програмування

Мова програмування описується поєднанням його семантики та синтаксису. Семантика дає нам значення кожної конструкції, яка можлива в цій мові програмування. Синтаксис дає нам свою структуру. Існує багато різних способів описати семантику мови програмування; однак, після десятиліть вивчення, існує переважно одна технологія для опису його синтаксису і полягає вона в описі синтаксису у вигляді певної граматики.

Для цього використовують переважно контекстно-залежні граматики відповідно до ієрархії Чомскі.

Ідея формальних мов полягає в виведенні математичних описів даних мов.

Формальну мову можна описати за допомогою наступних складових:

— термінальний символ – елементарна складова мови;

— нетермінальний символ – допоміжний символ для формування слів та речень;

— слово – набір символів(не обов'язково унікальних), що може існувати в рамках мови (позначається як $\sum^*$);

— алфавіт – повний набір унікальних символів мови (позначається як $\sum$);

— граматика – набір правил для формування слів мови.

Можна записати граматику для мови в наступному вигляді: ***G* = (*V*, T, *S*, *P*),** де V − скінченна непорожня множина, яку називають алфавітом мови, T − її підмножина, елементи якої називають термінальними (основними) символами, S − початковий символ, P − скінченна множина продукцій (або правил перетворення). Також можливий наступна форма запису граматики: ***G* = (*N*,**

Σ, *P, S*), де N – множина нетермінальних символів, Σ – множина термінальних символів, P – множина правил виводу, S – початковий символ [12].

Граматики типу 0 включають усі формальні граматики, тому мають назву - необмежені. Граматична мова типу 0 описується машиною Тюринга. Ці мови також відомі як рекурсивно перелічені мови.

Правила:

$\alpha \rightarrow \beta$

$\alpha \in V^* N V^*, \beta \in V^*$

Граматики типу 1 породжують мови, що залежать від контексту. Мова, породжена граматикою, розпізнається за допомогою лінійних обмежених автоматів.

Правила:

$\alpha A \beta \rightarrow \alpha'\gamma'\beta$

$A \in N, \alpha, \beta \in V^*, \gamma \in V+$

$S \rightarrow \varepsilon$ дозволене, коли серед правил P відсутнє

$\alpha \rightarrow \beta S\gamma$.

Граматики типу 2 породжують без контекстні мови. Мова, породжена граматикою, розпізнається недетермінованими автоматами з магазинною пам’яттю.

Правила:

$A \rightarrow \gamma$

$A \in N, \gamma \in V^*$

Граматики типу 3 породжують звичайні мови. Ці мови - це саме всі мови, які можуть бути прийняті автоматом скінченного стану.

Тип 3 є найбільш обмеженою формою граматики.

Правила:

$S \rightarrow \varepsilon$

$A \rightarrow aB$ (право лінійна)

$A \rightarrow Ba$ (ліво лінійна)

$A \rightarrow a$

$A \rightarrow \varepsilon$

$A, B \in N, a \in \Sigma$

Для написання мови програмування найкраще підійдуть правила побудови граматики 2-го та 3-го типу.

Центральною структурою даних у процесі побудови граматики для мови є сентенційна форма. Зазвичай його описують як рядок граматичних символів, а потім їх можна розглядати як представлення частково створеного програмного тексту. Однак для наших цілей ми також хочемо представляти синтаксичну структуру програми. Синтаксичну структуру можна додати при інтерпретації сентенційної форми як дерево, розташоване над сентенційною формою, так що листя дерева є граматичними символами. Це поєднання також називають деревом виводу [6].

Розглянемо для прикладу вираз ((2 + 2) * 2)

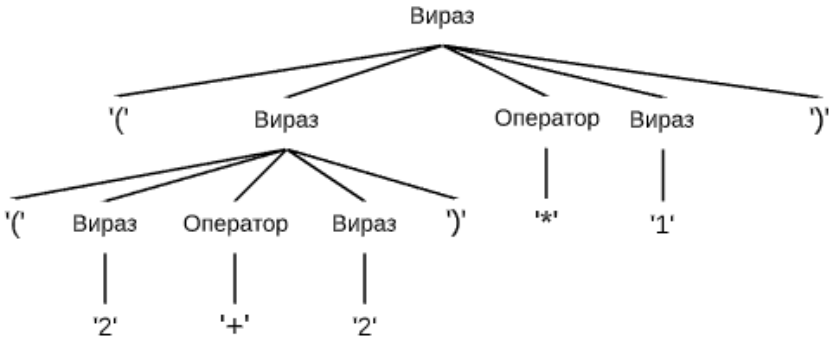

Рисунок 2.2 – Дерево виводу

Основною формою запису граматики є BNF – нотація Бакуса-Наура, або частіше – EBNF – розширена нотація Бакуса-Наура, яка передусім існує у вигляді стандарту - ISO-14977.

Для прикладу можна розглянути наступні правила:

letter = "A" | "B" | "C" | "D" | "E" | "F"

| "G" | "H" | "I" | "J" | "K" | "L" | "M"

| "N" | "O" | "P" | "Q" | "R" | "S" | "T"

| "U" | "V" | "W" | "X" | "Y" | "Z";

vowel = "A" | "E" | "I" | "O" | "U";

consonant = letter - vowel;

Дана форма запису є досить придатною для вираження вкладених правил та рекурсії, але менш зручною для вираження повторень, хоча, звичайно, може виражати їх через рекурсію.

Розширена форма Backus-Naur (EBNF) - це сукупність розширень до форми Backus-Naur. Не всі вони суто суперсети, оскільки деякі змінюють відношення визначення правил :: = до =, а інші видаляють кутові дужки з нетерміналів.

Більш важливими, ніж незначні синтаксичні відмінності між формами EBNF, є додаткові операції, які він дозволяє виконувати в розширеннях.

У EBNF квадратні дужки навколо виразу [A] вказують, що цей вираз необов'язковий. Наприклад, правило: <term> :: = ["-"] <factor>, дозволяє факторам бути як позитивними так і негативними.

Фігурні дужки У EBNF вказують на те, що вираз може бути повторений нуль або більше разів. Наприклад, правило: <args> :: = <ar> {"," <arg>}, визначає список аргументів, розділений комами.

Для позначення пріоритетності граматики EBNF можуть використовувати дужки (), щоб визначити групу виразу. Наприклад, правило: <expr> :: = <term> ("+" | "-") <expr>, визначає форму вираження, яка дозволяє як додавання, так і віднімання.

У деяких формах EBNF оператор «,» позначає конкатенацію, а не покладається на протиставлення.

Існує ряд властивостей граматики та її компонентів, які використовуються при обговоренні:

1) Ліва та права рекурсивність;
2) Нульові нетермінали;
3) Неоднозначність.

Нетермінальний N є рекурсивним ліворуч, якщо, починаючи з сентенційної форми N, ми можемо виробляти іншу сентенційної форму, починаючи з N. Прикладом прямої лівої рекурсії є вираз → вираз «+» слово | слово.

За визначенням граматика, яка містить одне або декілька ліво-рекурсивних правил, сама називається ліво-рекурсивною. Право-рекурсійні теж існують, але вони є менш важливим.

Нетермінальний N є нульовим, якщо, починаючи з сентенційної форми N, ми можемо створити порожню сентенційну форму ε. Граматичне правило для нульового нетерміналу називається ε -правилом. Важливо зауважити, що нульовість не повинна бути безпосередньо видною з ε -правила. Нетермінальний N безрезультатний, якщо він ніколи не може створити рядок символів терміналу: будь-яка спроба зробити це неминуче призводить до речення, яке знову містить N.

Приклад:

вираз → «+» вираз | «–» вираз

Граматика неоднозначна, якщо вона може виробляти два різних дерева виводу з однаковим листками в одному порядку. Це означає, що, коли ми втрачаємо дерево виводу через лінеаризацію тексту програми, ми не можемо однозначно його реконструювати; а оскільки семантика походить від дерева виводу, ми втрачаємо і семантику. Тому неоднозначних граматик слід уникати в специфікаціях мов програмування, де додана семантика відіграє важливу роль.

Граматика є однією з найважливіших складових будь-якої мови програмування, так як, саме вона формує синтаксис та семантику мови, та й взагалі її загальне сприйняття. Опис граматики як контекстно вільної є найкращим для мови програмування та за допомогою опису в форматі EBNF дозволяє покращити розуміння мови.

Використовуючи EBNF можна описати графічно програму будь-якої складності у вигляді дерева виводу, або ж сформувати подібне дерево

програмно при синтаксичному аналізі коду компілятором.

2.2 Лексичний аналіз

Початкова форма даних при компіляції являється потоком символів, що складають текст програми, і компілятор, як очікується, створить з нього проміжний код, що дозволить обробляти контекст і переводити в цільовий код. Це робиться, попередньо відновивши синтаксичну структуру програми шляхом розбору тексту програми відповідно до граматики мови. Оскільки значення програми визначено з точки зору її синтаксичної структури, володіння цією структурою дозволяє генерувати відповідний проміжний код.

Перший етап такого розбору тексту має назву – лексичний аналіз та полягає в виділенні серед загального потоку символів лексеми та описі їх у вигляді токенів. Лексемами у даному випадку виступають будь-які групи символів які формують слова відповідної граматики формальної мови. А їх токенізація – це групування таких слів під певні категорії та додавання певної інформації до кожної лексеми [5].

До лексем зазвичай належать:

— зарезервовані слова (if, else, for, while, class);
— ідентифікатори (int, string, bool);
— числові константи;
— літерні константи;
— оператори;
— допоміжні(нетермінальні) символи;
— коментарі.

Токени, при написанні компілятора, досить легко описати у вигляді класів і при аналізі тексту описувати кожну лексему як об’єкт відповідного класу.

Розглянемо наступний математичний вираз мовою C#:

```
int a = 2 + 2;
```

Для даного виразу достатньо виділити наступні типи токенів: ідентифікатор типу, змінна, оператор, числова константа.

Опишемо типи токенів мовою C#

Тоді лексеми виразу можна класифікувати наступним чином (приймаючи дільником лексем - пробіл):

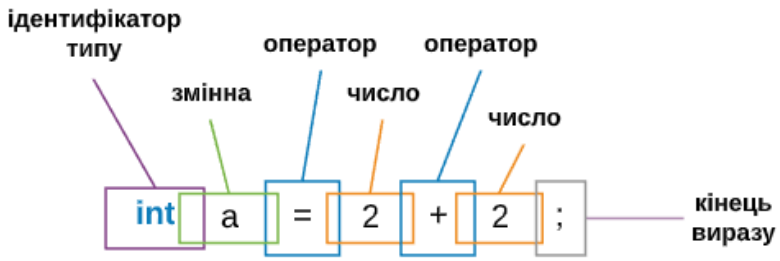

Рисунок 2.3 – Лексичний розбір виразу

А програмне уявлення такого розбору буде наступним:

```
Token[] tokens = new Token[]
    {
        new TypeName("int"),
        new Variable("a"),
        new Operator('='),
        new Number(2),
        new Operator('+'),
        new Number(2),
        new EndOfStatement()
    };
```

З такого виду запису можна помітити перехід від символьного потоку до набору (масиву) токенів. Після такого перетворення легко побудувати дерево відношення токенів, яке формує синтаксис та є задачею синтаксичного аналізу. Зазвичай процес лексичного аналізу при компіляції називається токенізацією а синтаксичний – парсингом, але часто ці 2 процеси поєднуються в одним – побудову синтаксичного дерева, проте між цими 2-ма процесами все ж є певна межа, яку важливо розуміти.

Маючи як лексичний, так і синтаксичний аналіз, потрібно вирішити, де лежить межа між цими двома. Лексичний аналіз виробляє лексеми, а синтаксичний аналіз споживає їх, але що саме є лексемою? Частина відповіді походить з визначення мови, а частина - з дизайну. Для визначення цього можна ввести наступне правило: «Якщо його можна відокремити від лівого та правого сусідів пробілом, не змінюючи значення, це знак; інакше це не так.» Якщо між двокрапкою та знаком рівності вводиться пробіл «: =» , це дві лексеми, і кожен повинен відображатися, відповідно до граматики, як окремий знак. Якщо вони повинні стояти поруч, не порушуючи при цьому правил заданої граматики - це один маркер. Це не означає, що маркери не можуть включати пробіл, для прикладу можна взяти сканування рядків, оскільки і вони є лексемами за вищезазначеним правилом: додаючи пробіл у рядковий літерал ми змінюємо його значення [3].

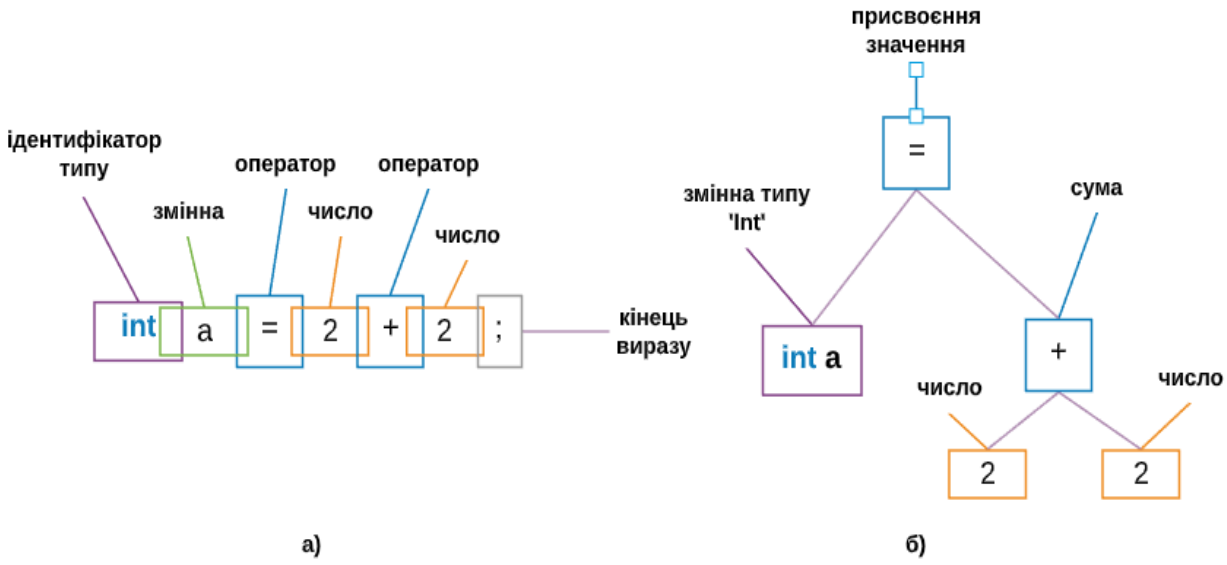

Рисунок 2.4 – а) - Лексичний розбір виразу; б) – Синтаксичний розбір виразу

Коментарі та пробіл не є лексемами, оскільки синтаксичний аналізатор їх не споживає. Вони, як правило, відкидаються лексичним аналізатором, але часто корисно їх зберегти, щоб показати текст програми, що оточує помилку. Також на практиці варто додавати назву файлу, номер рядка та позицію символу, в якій маркер знайдено (або власне там, де він почався). Така інформація

є неоціненною для подання зручних для користувача повідомлень про помилки, які можуть з’явитися набагато пізніше в компіляторі, коли власне текст програми може бути довго викинутий із пам'яті.

2.3 Синтаксичний аналіз

Синтаксичний аналіз - це друга фаза процесу проєктування компілятора, що настає після лексичного аналізу, і на відміну від нього, аналізується синтаксична структура виокремлених токенів. Синтаксичний аналізатор перевіряє, чи вказаний вхід відповідає коректному синтаксису заданої мови програмування. Перевірка всіх синтаксичних правил виконується за допомогою граматики, заданої раніше [4].

Граматики є важливим інструментом мовної специфікації; вони мають кілька важливих аспектів. По-перше, граматика служить для накладення структури на лінійну послідовність лексем, яка є програмою. Ця структура є найважливішою, оскільки семантика програми визначена з точки зору вузлів у цій структурі. Процес фіксації структури у потоці токенів називається розбором, а модуль, що виконує це завдання, є парсером. По-друге, використовуючи методи з поля формальних мов, аналізатор може бути побудований автоматично з граматики. Це чудова допомога в побудові компілятора. По-третє, граматики є потужним інструментом документації. Вони допомагають програмістам писати синтаксично правильні програми та дають відповіді на детальні запитання про синтаксис.

Кінцевою формою даних після аналізу є абстрактне синтаксичне дерево, або ж дерево парсингу. Сам процес аналізу теж часто називають парсингом.

Синтаксичний аналіз важливий не тільки формування синтаксичного дерева, яке згодом використовується для побудови проміжного коду, а й детальним виявленням синтаксичних помилок при компіляції.

Абстрактне синтаксичне дерево (англ. AST – Abstract Syntax Tree) – це представлення структури програми у вигляді дерева, де кожен вузол є виразом. При цьому, для такого дерева повинні виконуватись наступні правила:

— AST – не є строго бінарним деревом, можливі випадку унарних операторів з одним вузлом, або ж конструкції if/else з безліччю вузлів;

— AST – є гетерогенним деревом, тобто може містити вирази різного типу.

Розглянемо для прикладу побудову AST для конструкції розгалуження виразів if/else спільних для більшості мов програмування:

```
if (a % 2 == 0)
{
    Console.WriteLine("Even");
}
else
{
    Console.WriteLine("Odd");
}
```

Приведений запис можна зобразити у вигляді наступного абстрактного синтаксичного дерева:

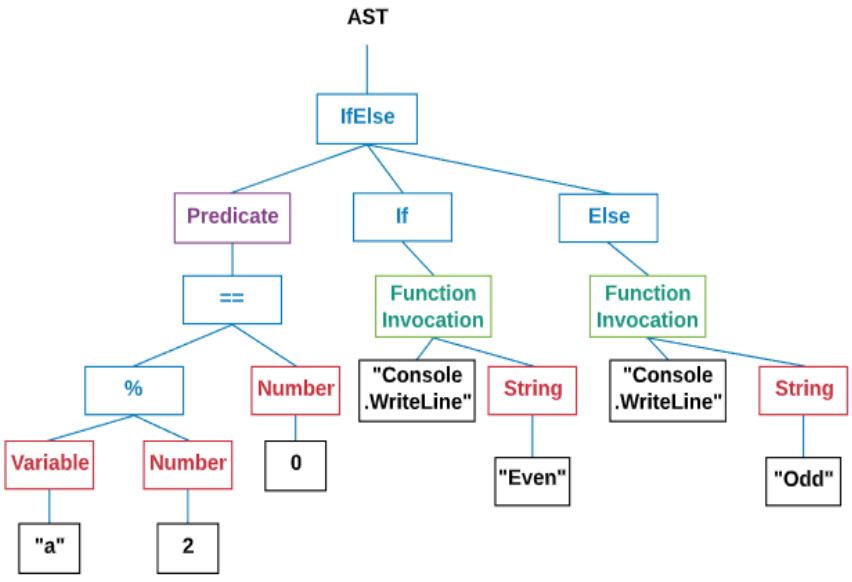

Рисунок 2.5 – Абстрактне синтаксичне дерево

Досить легко сформувати програмне представлення дерева для приведеної конструкції, користуючись статичними методами базового класу «Expr», який виступає загальним для визначення вузла дерева:

```
new AST(
        Expr.IfElse(
          Expr.Equal(
            Expr.Mod(
              Expr.Var("a"),
              Expr.Num(2)),
            Expr.Num(0)
            ),
          new[]
          {
            Expr.InvokeFunction("Console.WriteLine", Expr.Str("Even"))
          },
          new[]
          {
            Expr.InvokeFunction("Console.WriteLine", Expr.Str("Odd"))
          })
        );
```

Можна зазначити, що даний запис, навіть описаний програмно, завдяки форматуванню та вкладеності викликів, дуже нагадує дерево.

Метод парсингу будує синтаксичне дерево для заданої послідовності лексем. Побудова синтаксичного дерева означає, як вже було показано попередньо, що має бути створено дерево з вузлами певної структури і ці вузли повинні бути позначені граматичними символами таким чином, щоб:

— вузли-листки позначені як термінали, а внутрішні як нетермінали;

— вершина дерева позначена старт-символом заданої граматики;

— термінали, що позначають вузли листів, відповідають послідовності

лексем у тому ж порядку, що вони трапляються на вході.

Існує два основні способи парсингу: зверху вниз та знизу вгору. Для парсерів типу зверху вниз можна вибрати написання вручну або автоматичне генерування, але парсер типу знизу вгору може бути лише згенерований.

Розбір зліва направо починається з перших декількох лексем вхідного сигналу та синтаксичного дерева, яке спочатку складається лише з верхнього вузла. Верхній вузол позначений символом початку.

Методи синтаксичного розбору можна виділити за порядком, в якому вони будують вузли в синтаксичному дереві: метод зверху вниз конструює їх в прямому (preorder) порядку, методи знизу вгору в зворотному (postorder). Спосіб прямого починається вгорі і будує дерево вниз, щоб відповідати лексемам на вході; метод зворотного порядку поєднує лексеми на вході в частини дерева, щоб остаточно сконструювати верхній вузол. Два способи роблять зовсім інші речі, коли вони конструюють вузол.

Існує також центральний порядок, але він рідко застосовується для обходу абстрактних синтаксичних дерев.

Процес обходу починається вгорі дерева в обох випадках і врешті-решт відвідує всі вузли на дереві; Однак порядок відвідування вузлів відрізняється. Під час обходу вузла N в прямому порядку, процес вперше відвідує вузол N, а потім переміщує N підрядок у порядку зліва направо. Під час обходу вузла N в порядку зворотного процес вперше перетинає підряди N у порядку зліва направо, а потім відвідує вузол N. Інші варіанти (багаторазові відвідування, змішування відвідувань усередині проходу зліва направо, відхилення від обхід зліва направо) можливі, але менш звичний.

Варто зауважити, що тут беруть участь три різні поняття: відвідування вузла, що означає виконання певної операції над вузлом, яка є значущою для загального алгоритму, за допомогою якого виконується обхід; обхід вузла, що означає відвідування цього вузла та обхід його входжень у певному порядку; і обхід дерева, що означає перехід від його верхнього вузла рекурсивно до всіх інших. «Відвідування» належить до алгоритму; «Проходження» в обох

значеннях належить до механізму управління. Це розділяє дві проблеми і є джерелом корисності концепції обходу дерев. Зазвичай ці терміни часто плутають.

Процес парсингу зверху вниз починається з побудови верхнього вузла дерева, який, як відомо, позначається символом початку. Після цього він будує вузли в синтаксичному дереві на рівень нижче, що означає, що верхня частина піддерева побудована до того, як будь-який з його нижніх вузлів.

Коли аналізатор згори вниз будує вузол, мітка самого вузла вже відома, скажімо, N; це справедливо для верхнього вузла, і ми побачимо, що це справедливо і для всіх інших вузлів. Використовуючи інформацію з вхідних даних, аналізатор визначає правильну альтернативу для N. Знаючи, яка альтернатива застосовується, вона знає мітки всіх дітей цього вузла з позначкою N. Парсер потім приступає до побудови першої дитини N. Процес визначення правильної альтернативи для самої лівої дитини повторюється на подальших рівнях, поки не буде побудована крайня ліва дитина, яка є символом терміналу. Потім термінал «відповідає» першому токену в програмі.

Це не відбувається випадково: аналізатор згори вниз вибирає альтернативи вищих вузлів саме так, щоб це відбулося. Тепер ми знаємо «чому перший токен існує», який сегмент синтаксичного дерева створив перший токен. Потім аналізатор залишає термінал позаду і продовжує, будуючи наступний вузол в прямому порядку.

Дивіться рисунок 2.6, на якому велика крапка - це вузол, який будується, менші крапки - це вузли, які вже були побудовані, а порожні точки вказують на вузли, мітки яких уже відомі, але ще не побудовані. Про решту дерева розбору ще нічого не відомо, тому ця частина не показана. Підсумовуючи це, головне завдання аналізатора зверху вниз - вибрати правильні альтернативи для відомих нетерміналів.

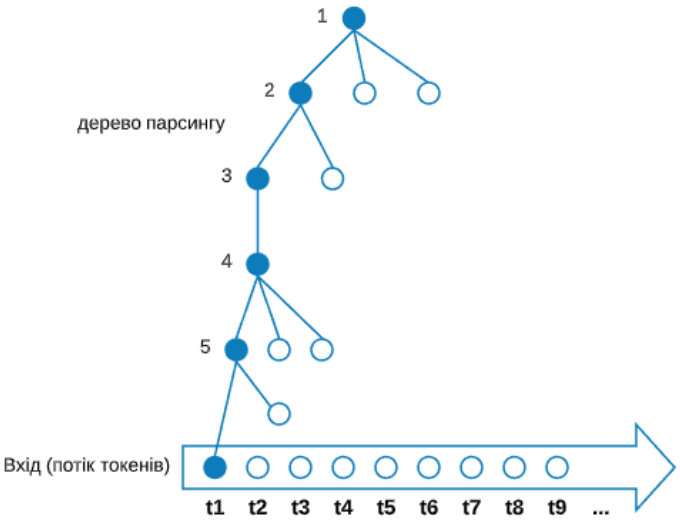

Рисунок 2.6 – Парсинг зверху-вниз

Метод розбору знизу вгору конструює вузли в дереві синтаксису в «післяпорядку»: верхівка піддерева будується після того, як були побудовані всі його нижні вузли. Коли аналізатор знизу вгору будує вузол, усі його діти вже побудовані, і вони є присутніми та відомими; мітка самого вузла також відома. Потім аналізатор створює вузол, позначає його і підключає його до своїх дітей.

Аналізатор знизу вгору завжди створює вузол, який є вершиною першого повного піддерева, з яким він зустрічається, коли проходить зліва направо через вхід; повне піддерево – це дерево, усі діти якого вже побудовані.

Токени ж розглядаються як піддерева висоти 1 і будуються по мірі їх зустрічі. Зрозуміло, нове піддерево має бути вибране таким чином, щоб воно було піддією дерева розбору, але очевидною проблемою є те, що ми ще не знаємо дерева розбору. Діти першого піддерева, яке буде побудовано, мають лише вузли листки, позначені терміналами, і правильна альтернатива вузла обрана для їх узгодження. Далі, у другому піддереві у вході знайдено всіх дітей, чиї діти вже побудовані; діти цього вузла тепер можуть залучати нелистові вузли, створені попередньою побудовою вузлів. Для нього побудований вузол, з міткою та відповідною альтернативою. Цей процес повторюється, поки остаточно не будуть побудовані всі діти верхнього вузла, після чого побудується сам верхній вузол і не буде завершено аналіз.

На рисунку 2.6 представлений аналізатор (розпізнавання) його першого, другого та третього вузлів. Зафарбована крапка знову вказує на побудований вузол, не зафарбована - на не побудований. Перший вузол охоплює лексеми t3, t4 і t5; другий лексеми t7 і t8; а третій вузол охоплює перший вузол, маркер t6 та другий вузол. Ще нічого не відомо про існування інших вузлів, але гілки були намальовані вгору від токенів t1 і t2, оскільки ми знаємо, що це не може бути частиною вже побудованого піддерева, інакше це піддерево було б першим, що будується. Підводячи підсумок, головне завдання аналізатора знизу вгору - це багаторазово знаходити перший вузол, всі діти якого вже побудовані.

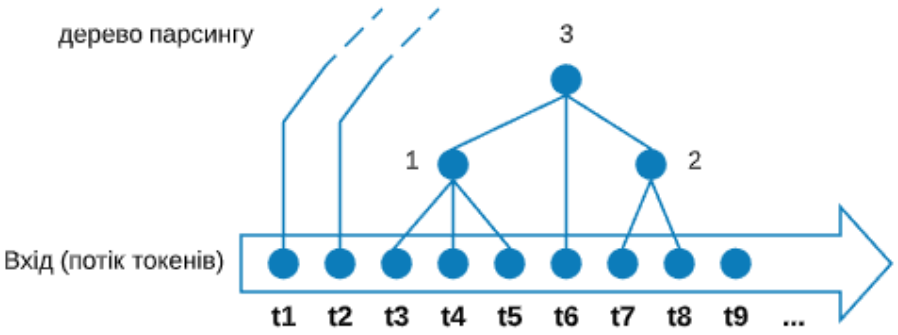

Рисунок 2.7 – Парсинг знизу-вгору

Відносно новим підходом до написання програм парсингу тексту з досить складними правилами граматики є побудова комбінаторів парсерів.

Цей підхід зародився у функціональному програмуванні та ставши досить популярним полягає в побудові рекурсивних парсерів у вигляді функцій вищого порядку (комбінаторів), які описують граматичні конструкції аналізованого тексту. При цьому функцією вищого порядку вважається будь-яка функція, яка приймає в якості параметра, або ж повертає іншу функцію.

Такі парсери екземпляри монади, алгебраїчної структури з математики, що є досить корисною для вирішення ряду обчислювальних проблем. У функціональному програмуванні монада - це мовно конструкція, яка дозволяє

загально структурувати програми. Монади досягають цього шляхом надання власного типу даних (певного типу для кожного типу монади), який представляє конкретну форму обчислень, поряд із однією процедурою обертання значень будь-якого базового типу всередині монади (отримання монади ного значення) та іншої складати функції, які виводять монахині значення (називаються монади ними функціями) [7].

Взявши за основу ці визначення ми можемо описати парсер як функцію яка приймає іншу функцію, що описує правило граматики, а монадою, або ж комбінаторами в цьому випадку будуть інші функцію, які комбінуються парсери між собою. Тобто, за допомогою даного підходу, ми можемо створювати добутки, суми, повторення та інші комбінації правил.

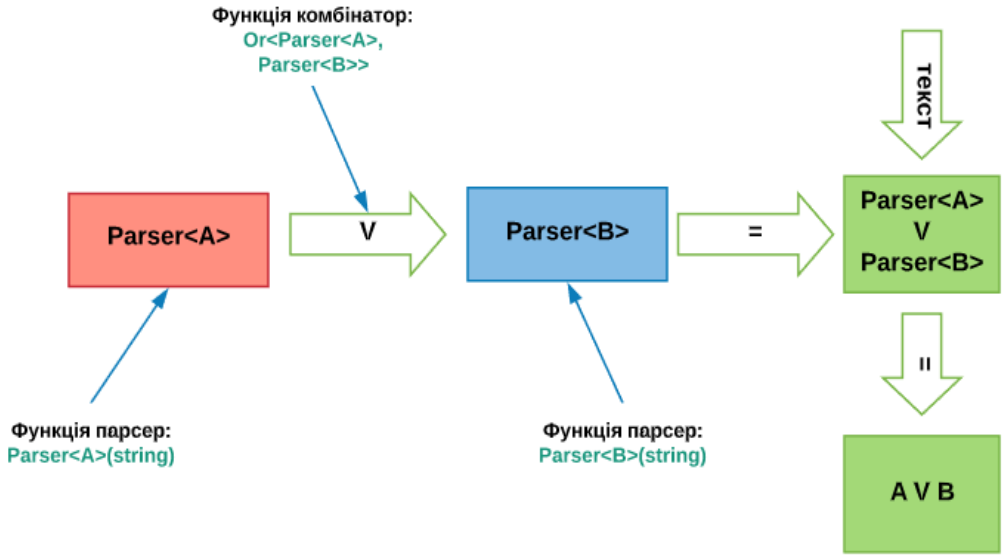

Рисунок 2.8 – Кон’юнкція парсерів

Представивши найпростіші правила граматики у вигляді парсерів, їх можна легко комбінувати в більш складні за допомогою різних операцій.

Найбільш поширеними є:

— повторення – для лексем, які можуть зустрічатися декілька разів підряд;

— кон’юнкція – для визначення парсеру для 2-х і більше різних лексем;

— lazy обробка для рекурсивно вкладених виразів (Наприклад: уявленні вкладених математичних виразів в дужках, де ліва або права частина є або виразом або простим числом);

— заперечення – для парсингу всі інших символів крім зазначеної лексеми;

— опціональність – для лексем які є необов'язковими;

— вибір з багатьох – визначення кон'юнкції на наборі парсерів.

Ключовим в даному випадку є те, що кожен наступний парсер застосовує попередній, таким чином, базуючись на простих правилах, можна легко перейти до відносно складних синтаксичних конструкцій [8].

Для реалізації процесу парсингу в компіляторі TinyLang використовуються монадні комбінатори парсерів з бібліотеки LanguageExt.Parsec, що надає досить декларативний варіант синтаксису для їх побудови. Основною перевагою бібліотека є використання синтаксису LINQ, що дозволяє легко описувати парсери для досить складних синтаксичних конструкцій та вдало їх комбінувати.

Розглянемо для прикладу реалізацію парсеру для циклу For

```
public static Parser<Expr> For(Parser<Expr> parser)
{
    return from f in StrValue("for")
           from s in spaces
           from args in TokenParser.Parens(ForArgs(parser))
           from body in ScopeOrSingle(parser)
           select ForExpr.Define(args.Start, args.End, args.Step, body as Scope);
}

private static Parser<(Expr Start, Expr End, Option<Expr> Step)> ForArgs(Parser<Expr> parser)
{
    return from start in parser
           from s1 in spaces
           from terminal1 in StrValue("to")
           from s2 in spaces
           from end in parser
           from step in optional(Step(parser))
           select (start, end, step);
}

private static Parser<Expr> Step(Parser<Expr> parser)
{
    return from terminal1 in StrValue("step")
           from s in spaces
           from step in parser
           select step;
}
```

Парсинг ідентифікатора For

Рисунок 2.9 – Опис парсерів для синтаксичної конструкції «For»

В даному випадку парсинг конструкції розбивається на 3 основні частини:

— парсинг кроку циклу «step»;

— парсинг аргументів циклу;

— повний парсинг конструкції «for».

Кожна наступна синтаксична конструкція використовує попередню, що забезпечує більш просту реалізацію.

2.4 Обробка контексту

Лексичний і синтаксичний аналіз описаний в розділах 2.2 і 2.3, застосовується до тексту програми, формують абстрактне синтаксичне дерево (AST) з мінімальною, але. важливою мірою анотації: атрибути, надані лексичним аналізатором як початкові атрибути терміналів у листкових вузлах AST. Наприклад, токен, що представляє ціле число, має клас «integer», і його значення походить від подання лексеми; маркер, що представляє ідентифікатор, має клас «identifier», але для завершення подальших атрибутів може знадобитися час, поки механізм ідентифікації виконає свою роботу.

Лексичний аналіз та парсинг разом виконують безконтекстну обробку вхідної програми, це означає, що вони аналізують та перевіряють функції, які можна аналізувати та перевіряти локально або через рекурсивне вкладення. Інші функції, наприклад перевірка кількості параметрів у виклик функції проти кількості параметрів у його декларації, не підпадають під цю категорію. Вони вимагають встановлення та перевірки взаємних зв'язків, що є сферою обробки контексту [4].

Обробка контексту необхідна для двох різних цілей: для збору інформації для семантичної обробки та для перевірки контекстних умов, накладених мовною специфікацією.

Дані, необхідні для цих аналізів та перевірок, зберігаються як атрибути

у вузлах AST. Незалежно від того, чи вони фізично зберігаються там, чи фактично перебувають в іншому місці, наприклад, у таблиці символів або навіть у локальних змінних методу аналізу, вони є досить важливими, хоча зручність та ефективність можуть, звичайно, диктувати одну реалізацію або іншу.

Фаза обробки контексту виконує своє завдання, обчислюючи всі атрибути і перевіряючи всі умови контексту. Як і у випадку з аналізаторами, ви можете написати код для фази обробки контексту вручну або створити його на більш високому рівні. Найбільш звичайною формою специфікації вищого рівня є граматика атрибутів. Однак використання граматики атрибутів для обробки контексту набагато менш широко поширене, ніж граматики без контексту для обробки синтаксису: модулі обробки контексту все ще часто пишуться вручну. Дві можливі причини, чому це так, приходять на думку. По-перше, граматики атрибутів засновані на «парадигмі даних» програмування, парадигмі, в якій значення можна обчислити по суті в довільному порядку, за умови, що вхідні значення, необхідні для їх обчислень, вже були обчислені.

Тим не менш, граматики атрибутів дозволяють залишатися набагато ближче до умов контексту, що досить важливо при побудові компіляторів для багатьох сучасних мов програмування, наприклад, C ++, Ada та Java, оскільки ці мови мають великі та часто повторювані набори контекстних умов, які потрібно ретельно перевіряти.

Обчислення, необхідні для обробки контексту, можна додати у вільну контекстну граматику, яка вже використовувалась під час парсингу, у вигляді граматики атрибутів. Для вираження цих обчислень контекстну граматику розширюють двома функціями, однією для даних та однією для обчислень:

Для кожного граматичного символу S, кінцевого або нетермінального терміну встановлюються нульові або більше атрибутів, кожен з іменем та типом, як поля у записі; це формальні атрибути, оскільки, як і формальні параметри, вони складаються лише з імені та типу. Місце для фактичних атрибутів виділяється автоматично у кожному вузлі, створеному для S у абстрактному синтаксичному дереві. Атрибути використовуються для зберігання інформації про семантику, приєднану до цього конкретного вузла.

З кожним правилом виводу N → M1 ... Mn пов'язаний набір правил обчислень - правила оцінки атрибутів - які виражають деякі значення атрибутів лівої частини N та членів правої частини Mi в умови інших значень цих атрибутів. Ці правила оцінки також перевіряють контекстні умови та видають попередження та повідомлення про помилки [13].

Створені таким чином атрибути повинні задовольняти наступну умову: Атрибути кожного граматичного символу N поділяються на дві групи, які називаються синтезованими атрибутами та успадкованими атрибутами; правила оцінки всіх правил виводу N можуть розраховувати на значення успадкованих атрибутів N, що встановлюються батьківським вузлом, і самі зобов'язані встановлювати синтезовані атрибути N так, що вимога стосується граматичних символів, а не правила виводу.

Визначення таких атрибутів проводиться за допомогою збереження даних до таблиці символів завчасно, або ж в ході обходу дерева. Таким чином зберігаються та встановлюються типи даних, імена та значення змінних або ж констант, сигнатури функцій та інша інформація специфічна для кожного повторного запуску компіляції коду [11].

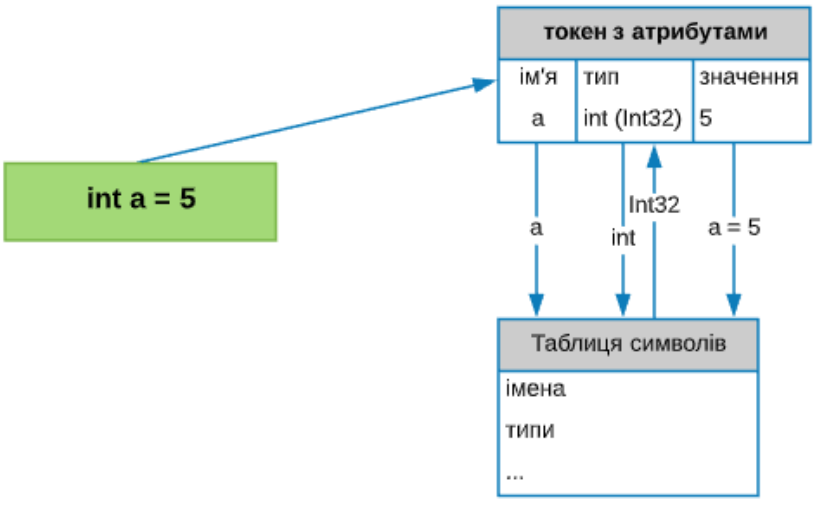

Рисунок 2.10 – визначення атрибутів контекстної граматики за допомогою таблиці символів

Варто зазначити, що таблиці символів, досить часто, розмежовують на певні рівні. Одна з можливих реалізацій може мати наступний вигляд :

— Глобальний рівень;

— Рівень простору імен;

— Рівень тіла методу.

Такий підхід дозволяє чітко розмежувати рівні доступу до імен змінних, або типів, а також дозволяє підтримувати унікальність імені лише на конкретному рівні, при чому імена змінних одного степеню вкладеності різних рівнів можуть повторюватись.

Пошук потрібного значення для атрибуту може мати певну ієрархічну структуру, починаючись від більш конкретних рівнів та закінчуючи більш глобальними, для уникнення конфліктів унікальності ідентифікаторів.

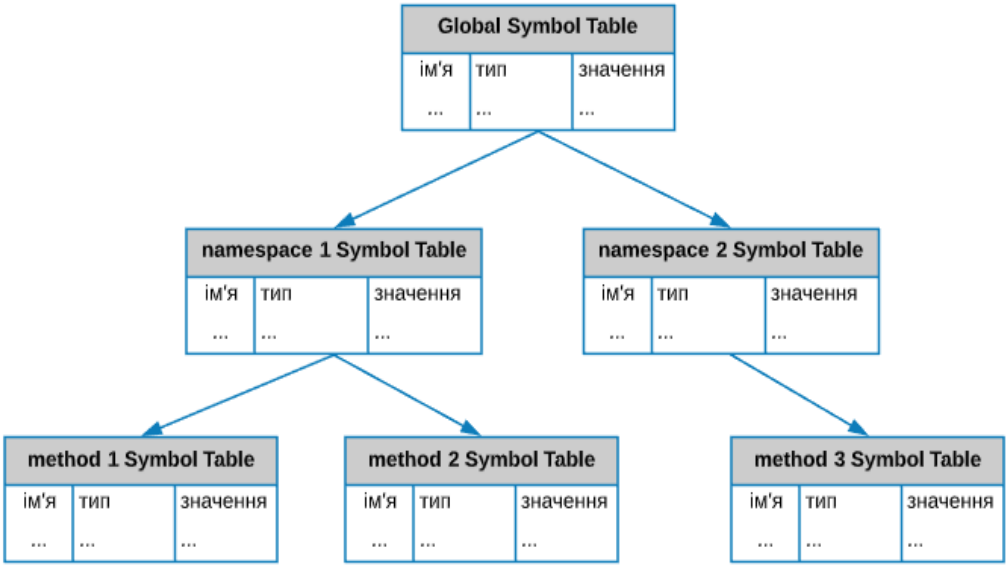

Рисунок 2.11 – Ієрархія таблиць символів

Якщо компілятор повинен обробляти невеликий обсяг даних, тоді таблиця символів може бути реалізована у вигляді невпорядкованого списку, який легко кодувати, але він підходить тільки для невеликих таблиць. Таблиця символів може бути реалізована одним із таких способів:

— Лінійний список;

— Бінарне дерево пошуку;

— Хеш-таблиця.

Серед всіх таблиць символів, більшість в основному реалізовані у вигляді хеш-таблиць, де сам символ вихідного коду розглядається як ключ для хеш-функції, а повертається значення є інформацією про символ.

2.5 Формування проміжного коду

Проміжний код використовується для перекладу вхідного коду в машинний код. Проміжний код лежить між мовою високого рівня та машинною мовою. Він використовується на останньому етапі верхнього рівня компіляції та є наступним виглядом вхідної програми після AST. Така імплементація обумовлена наступними перевагами та особливостями проміжного коду:

— якщо компілятор безпосередньо переводить вихідний код у машинний код без генерування проміжного коду, то для кожної нової машини потрібен повний нативний компілятор;

— проміжний код зберігає частину аналізу однаковою для всіх компіляторів, тому не потрібен повний компілятор для кожної унікальної машини;

— генератор проміжного коду отримує вхід від попередньої фази та фази семантичного аналізатора. Він приймає дані у вигляді анотованого синтаксичного дерева;

— за допомогою проміжного коду змінюється друга фаза синтезу компілятора відповідно до цільової машини.

Можна виділити 2 основні групи проміжного коду: високорівневий та низькорівневий.

Проміжний код високого рівня може бути представлений як вхідний код. Для підвищення продуктивності вхідного коду ми можемо легко застосувати модифікацію коду. Але для оптимізації цільової машини менш являється менш придатним.

Розглянемо різні варіанти представлення конструкції If/Else:

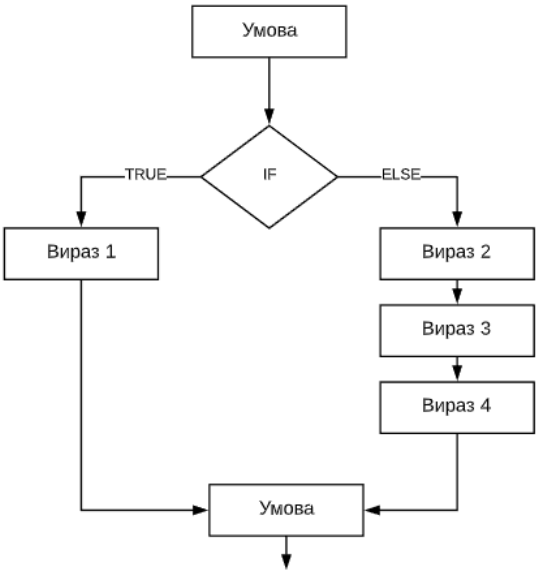

Рисунок 2.12 – Алгоритм конструкції If/Else

Токени збережені у вигляді масиву

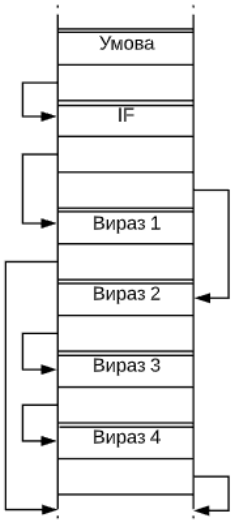

Рисунок 2.13 – Токени AST конструкції If/Else збережені у вигляді масиву

Токени у вигляді псевдо інструкцій проміжного коду

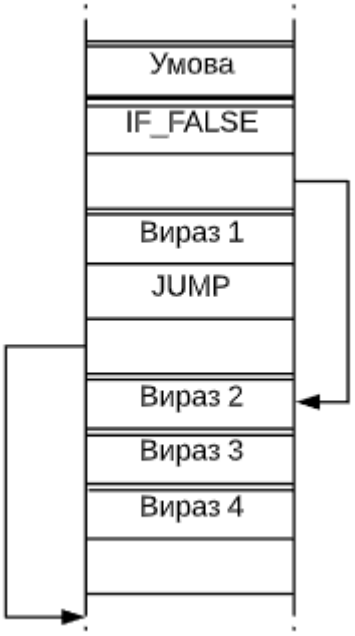

Рисунок 2.14 – Конструкція If/Else у вигляді псевдо інструкцій

Проміжний код низького рівня близький до цільової машини, що робить його придатним для розподілу реєстру та пам'яті тощо. Він використовується для залежних від машин оптимізацій. Зазвичай такий проміжний код виглядає досить схожим на інструкції ассемблеру та є більш популярною імплементацією. Найпопулярнішими представниками є Java Bytecode та .NET CIL.

Платформа .NET підтримує досить багато мов. У різних користувачів є різні конфігурації машин і різні операційні системи, невідомі для платформи. У цьому полягають головні проблеми компіляції .NET-сумісних мов. Щоб уникнути цього, Microsoft створює код, який називається IL-код або ж CIL (Common Intermediate Language).

CIL є проміжним кодом для всієї платформи .NET та надає наступні переваги:

— пришвидшує написання компіляторів для нових мов програмування;

— пришвидшує написання компіляторів для різних операційних систем;

— оптимізує код відповідно до платформи;

— додає підтримку BCL (Base Class Library) мовам всередині платформи, що дозволяє їм бути сумісними між собою;

— додає оптимізації синтаксичних конструкцій на етапі генерації проміжного коду.

Під час виконання компілятор з'ясовує, тип операційної системи, конфігурацію обладнання та навіть розрядність процесора та компілює оптимальний код відповідно до цього середовища.

Обробка IL відбувається за допомогою JIT (Just-In-time) компілятора, який рухається покроково відповідно до кожної інструкції IL та перетворює її в машинну [1].

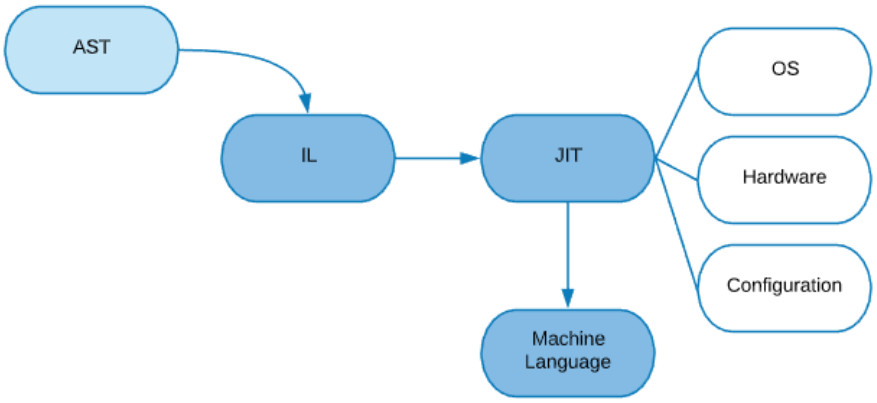

Рисунок 2.15 – Процес JIT-компіляції IL

Приклад If/Else описаного за допомогою IL інструкцій:

```
assembly extern mscorlib {}
.assembly Test
{
	.ver 1:0:1:0
}
	.module test.exe
	.method static void main() cil managed
{
	.maxstack 2
	.entrypoint
	ldc.i4 2
```

```
    ldc.i4 2
    beq Equal
      ldstr "Not equal"
      call void [mscorlib]System.Console::WriteLine (string)
    br Exit
    Equal:
      ldstr "Equal"
      call void [mscorlib]System.Console::WriteLine (string)
    Exit:
      Ret
}
```

Можна помітити, що дана реалізація за допомогою міток досить схожа на описану вище у вигляді псевдокоду.

Застосування проміжного коду при написанні компіляторів забезпечує більшу гнучкість, які зі сторони вхідного синтаксису, так і в плані згенерованих інструкцій машинного коду. Використовуючи JIT компіляцію, можна досягти генерації машинних інструкцій покроково, що дозволяє враховувати специфіку платформи на кожному кроці.

У цьому розділі було спроєктовано всі функціональні частини компілятору. Було впроваджено можливості комбінаторів парсерів за допомогою бібліотеки LanguegeExt.Parsec. Використовуючи провідні підходи до розробки та оптимізації компілятора, вдалося досягти значних результатів швидкодії компіляції програм мовою TinyLang. Згенеровані конструкції вихідного коду IL також оптимізовані та виконуються досить швидко. Завдяки цьому компілятор повністю готовий до використання в якості консольного застосунку та може виконувати інструкції мовою TinyLang, читаючи їх з зазначеного з файлу.

3 СПЕЦИФІКАЦІЯ СФОРМОВАНОЇ МОВИ ПРОГРАМУВАННЯ

В ході дипломного проєктування було сформовано новий синтаксис мови загального призначення TinyLang, а також створено компілятор та інтегроване середовище розробника. Сформована мова забезпечує основні синтаксичні конструкції та можливості для створення прикладних програм. Середовище розробника та компілятор дозволяють писати та виконувати застосунки мовою TinyLang.

3.1 Загальний опис, визначення граматики

TinyLang – мова, загального призначення, яку можна класифікувати як мову 3GL та 4GL покоління з переважаючими процедурними та функціональними синтаксичними конструкціями. Незважаючи на це, завдяки використанню .NET в якості платформи мова має таку ж систему типів як в об’єктно-орієнтованому C#.

Можна виділити наступні синтаксичні можливості мови:

— виділення примітивних типів даних (таких як int, str, bool);
— визначення змінних;
— визначення власних типів;
— визначення та виклик функцій;
— підтримка стандартних глобальних функцій (print);
— підтримка математичних операторів;
— підтримка булевих операторів;
— тернарний оператор;
— підтримка Лямбда-функцій;
— підтримка конструкцій розгалуження;
— підтримка циклів;
— вивід типів.

Мова також містить наступні зарезервовані слова: int, str, bool, func, type,

if, else, elif, for, in, to, new, while, do, return.

А також ряд операторів: «+», «–», «*», «/», «?», «:», «=>», «–>», «&&», «=», «==», «!», «!=», «<», «<=», «>», «>=», «||».

Дані конструкції описуються за допомогою контекстно-вільної граматики та можуть бути представлені у вигляді таблиці з правилами виводу формату EBNF – Додаток А.

Програмно – всі правила граматики записані у вигляді комбінаторів парсерів за допомогою функціонального розширення мови C# Language-Ext/Parsec та завдяки вбудованому синтаксису LINQ виглядають досить декларативно та дещо нагадують відповідні їм правила виводу.

Після цього модуль генерації коду рекурсивно проходить дерево та формує вихідні конструкції мовою IL та збирає їх в .dll файл або ж образу виконує. Крім цього є також окрема бібліотека для підключення мови в якості DSL у інші мови платформи .Net. Для цього потрібно підключити nuget пакет TinyLang.Fluent та додати посилання на нього з проєкту. Після імпорту простору імен по замовчуванню: TinyLang.Fluent, у файли з’явиться доступ до основного класу бібліотеки – TinyLangEngine, який є, свого роду, «класом-будівельником» для тексту програми мовою TinyLang.

Використання такого розширення виглядає наступним чином:

```
// 1) lambda functions
    var lambdaEx = @"
            add = (a: int, b: int) => a + b
            mul = (a: int, b: int) => a * b
            func binaryOp(a: int, b: int, op: (int, int)->int) => op(a, b)
            addRes = binaryOp(3, 2, add)
            mulRes = binaryOp(3, 2, mul)
            printF(""3 + 2 = {0}"", addRes)
            printF(""3 * 2 = {0}"", mulRes)
        ";
using var engine = TinyLangEngine
            .FromScript(lambdaEx)
```

```
        .AddStatement(st => st.Print("Executed by TinyLangEngine"));
engine.Execute(out var ast);
```

Початковий скрипт передається у вигляді стрічки в метод «FromScript()» класу TinyLangEngine, який є ключовим для бібліотеки та загальної взаємодії з кодом TinyLang. Наступна стрічка виклику методу «AddStatement(…)» додасть до кінця скрипту виклик функції виводу в консоль. Остання ж стрічка виконує згенерований код та містить AST в якості додаткового вихідного параметра.

Такий підхід дозволяє не лише використовувати компілятор за своїм прямим призначенням, а й для написання різних IDE та розширень до існуючих текстових редакторів для підтримки синтаксису TinyLang. Використання бібліотеки дозволило досить легко написати IDE для TinyLang мовою C# та використовуючи графічні можливості платформи WPF. Завдяки використанню методу «Execute» з додатковим вихідним параметром AST – стали можливими такі функції інтегрованого середовища розробника як перегляд згенерованого в ході компіляції абстрактного синтаксичного дерева та його експорт в форматі JSON.

3.2 Основні можливості

Розглянемо основні синтаксичні можливості TinyLang на конкретних прикладах, а також порівняємо їх з аналогічними у вже існуючих мовах програмування. Хоча на даному етапі свого розвитку TinyLang не надає повний перелік можливостей спроєктованих граматикою, але має повний функціонал для написання програм.

Основною можливістю будь-якої мови програмування є об’явлення та присвоєння значення змінним

В TinyLang, на відміну від інших мов програмування, використання будь-якого не зарезервованого слова в контексті присвоєння йому значення є водночас і декларацією і присвоєнням.

```
x = 2
y = 4
```

Проте використання ідентифікатора, якому не було присвоєне значення раніше викличе помилку під час компіляції. При цьому важливим є момент типізації, при визначенні змінної тип не вказується, але наступні записи значення можливі лише початкового типу даних, тобто мова цілком статично типізована.

Будь-яка мова програмування також надає базові математичні можливості і TinyLang не є виключенням.

```
x = 2
y = 4
z = (2 * (x + y)) / 4
```

Компілятор здатний інтерпретувати вирази будь-якої вкладеності з врахуванням пріоритетності операторів.

В TinyLang, для представлення булевих значень, є окремий тип даних – bool, а також рад операторів для основних булевих перетворень. Крім того, для перевірки рівності використовуються оператори: «==» та «!=».

```
x = 2
y = 3
someEquation = ((x != y) || true) && false
```

Для представлення операторів розгалуження використовується набір зарезервованих ідентифікаторів: if, elif, else.

Розглянемо наступний приклад:

```
monthNumber = 7
msg = "Okay."
if((monthNumber > 2) && (monthNumber < 12) )
{
   msg = "Winter is coming,"
}
```

TinyLang також підтримує тернарний оператор, тому такий код легко може бути переписаний у вигляді виразу:

```
monthNumber = 7
msg = (monthNumber > 2) && (monthNumber < 12) ? "Winter is coming," :
"Okay."
```

Синтаксична конструкція циклу For дещо відрізняється від аналогічної в С-подібних мовах програмування. Замість звичних 3-х виразів (ініціалізації індексу, умови виходу та дії над індексом) розділених крапкою з комою в TinyLang використовується досить декларативний вид запису: «For … to … step …». Перший блок використовується для задавання початкового індексу, другий для кінцевого, останній ж блок «step» є необов'язковим та задає крок збільшення або зменшення індексу.

Розглянемо приклад:

```
for(i = 0 to 10 step 1)
{
   printF("{0} ", i)
}
```

Написаний код виводить в командну стрічку числа від 0 до 10 розділені пробілами. Код виглядає дещо простіше ніж наприклад аналогічний йому в С, проте, завдяки можливостям мови може бути спрощений ще більше та записаний навіть в одну стрічку:

```
for(0 to 10) printF("{0} ", index)
```

В даному випадку назва змінної індексації по циклу пропущена, тому для цього використовується зарезервоване слово «index», блок задання кроку теж пропущений та по замовчуванню використовується одиниця. Крім того, тілом циклу може бути не тільки блок, а й один вираз, тому фігурні дужки теж можна пропустити.

TinyLang також містить цикл «for each», який «пробігається» по всім елементам колекції, проте виглядає він відмінно від більшості мов та є дещо схожим на аналогічний у VisualBasic.

```
for(x in set)
{
   print(x)
}
```

Завдяки такій реалізації використовується лише одне ключове слово для описання 2-х різних типів циклів.

Цикли «while» та «do while» виглядають аналогічно більшості С-подібних мов:

```
while(false)
{
   print("never get there")
}
do
{
   print("infinity")
}
while(true)
```

Визначення функції синтаксисом TinyLang починається з ключового слова «func» та круглих дужок з переліченими аргументами зі специфікацією типів, а також містить блок тіла функції. Позначення типу аргументу виглядає наступним чином:

```
number: int
```

В даному випадку зліва іде ідентифікатор аргументу, а після назва типу. Також можливе визначення типу який повертає функція, але це є необов'язковим завдяки можливості виводу типів компілятора.

```
func CheckNumbers(left: int, right: int): str
{
   res = "equals"
   if(left > right)
```

```
    {
        res = "left is more than right"
    }
 elif(left < right)
    {
        res = "left is less than right"
    }
    return res
}
msg = CheckNumbers(2, 1)
print(msg)
```

Важливо зауважити, що функція «print» є вбудованою та використовується для виводу значення в консоль.

На відміну від більшості ООП мов, в TinyLang функції представляють окремий тип даних та можуть бути присвоєні змінній, або ж передані в якості аргументу в інші функції. Ця можливість дозволяє підвищити декларативність мови та наблизити її до функціональних та зменшити необхідність у визначенні окремих типів для зберігання в них функцій. та. передачі таких типів між іншими функціями.

```
add = (a: int, b: int) => a + b
mul = (a: int, b: int) => a * b
func binaryOp(a: int, b: int, op: (int, int)->int) => op(a, b)
addRes = binaryOp(3, 2, add)
mulRes = binaryOp(3, 2, mul)
printF("3 + 2 = {0}", addRes)
printF("3 * 2 = {0}", mulRes)
```

Серед ключових моментів можна виділити: визначення лямбда-функції: add = (a: int, b: int) => a + b, де зліва від оператора «=>» ідуть аргументи, а після нього іде вираз який буде виконуватись над аргументами; визначення аргументу з типом функції - (int, int)->int, який визначає сигнатуру функції яка

повинна бути підставлена параметром.

В даному випадку також використовується вбудована функція «printf», яка виводить в консоль форматовану стрічку.

На відміну від більшості ООП мов, в TinyLang не має такого поняття як клас або ж структура, є лише визначення типу. Типи не можуть наслідуватись, проте мають достатньо спрощений синтаксис, а також вже згенеровані методи їх порівняно по всім полям, а також приведення до текстового формату. Завдяки цьому функції виводу типів в консоль будуть виводити значення та імена всіх доступних полів типу.

Розглянемо наступну програму:

```
type User(name: str, age: int)
type ValidatedUserResult(user: User, isValidated: bool)
func valid(user: User) => new ValidatedUserResult(user, true)
func invalid(user: User) => new ValidatedUserResult(user, false)
func validateAge(user: User) => user.age > 18 ? valid(user) : invalid(user)
u = new User("Vlad", 21)
print(validateAge(u))
```

Завдяки автоматично-згенерованим функціям, виводом програми буде: ValidatedUserResult(user(User(name(Vlad), age(21))), isValidated(True)). Визначення типів в TinyLang дещо схоже на визначення функції, що скорочує варіативність синтаксичних конструкцій та полегшує вивчення мови. Після визначення типу, його ім’я стає доступним для позначення типів аргументів, тому створені типи можна використовувати як параметри або ж повертати з функцій. Створення конкретного типу імплементоване з використанням ключового слова «new», вже знайомим з інших мов платформи, таких як C#, або ж VisualBasic.

Також є ряд можливостей для яких не імплементовані алгоритми реалізації проміжного коду IL, але описані парсери для формування AST.

Досить популярною конструкцією серед функціональних мов програмування є можливість формування Алгебраїчні типи даних (ADT).

Данна синтаксична конструкція виражає представлення типів даних не в якості об’єктно-орієнтованих ієрархій наслідування, а в якості алгебраїчних композицій. Така конструкція лише набирає свою популярність, хоча й була застосована вперше досить давно.

В TinyLang її реалізація виглядає наступним чином:

```
type User =
Simple: (Age: int * Name: str) |
Admin: (Name: str)
```

Основними операціями є алгебраїчна сума – «*», яка виражає операцію «І» та алгебраїчний добуток – «|», який виражає операцію «АБО». За допомогою цього функціоналу можна досить декларативно описувати бізнес модель застосунку використовуючи мінімум коду.

Ще однією можливістю функціональних мов є поняття кортежів. Ідея кортежів полягає в зв’язуванні даних між собою без об’явлення додаткових типів, що допомагає значно економити час при моделюванні тимчасових складних об’єктів. TinyLang для таких цілей надає досить простий синтаксис:

```
a = 5
b = 10
func Sum(args: (int, int)) => args(1) + args(2)
args = (a, b)
s = Sum(args)
```

У функціональному програмування досить важливою є можливість часткового виклику функцій. Наприклад, при передачі в функцію від 3 аргументів лише одного аргументу, такий вид запису поверне нам іншу функцію від 2 аргументів, які залишилися.

Приклад:

```
func SumThreeNumbers(a: int, b: int, c: int) => a + b + c
sumTwoNumbers = SumThreeNumbers(1)
incrementSum = sumTwoNumers(2)
incrementSum(3)
```

Такий запис аналогічний наступному:

```
SumThreeNumbers(1, 2, 3)
```

Така синтаксична конструкція дозволяє зберігати частину аргументів, якщо інша ще не порахована.

Мови платформи .Net мають підтримку потужного функціоналу ітераторів, який базується на інтерфейсах IEnumerable та IEnumerator. Зазвичай генерування послідовностей полягає у приведенні колекцій до інтерфейсу IEnumerable<T>, або ж використання оператору yield. На відміну від інших мов платформи, TinyLang дозволяє повертати послідовність не використовуючи стандартні колекції або ж додаткові оператори.

Послідовність однотипних даних можна просто перелічити через кому використовуючи наступний запис:

```
seq(1, 2, 3, 4, 5)
```

Це значно зменшує кількість коду та покращує декларативність запису послідовностей.

Будь-яка мова програмування потребує певної взаємодії типів та функцій над ними. Данна конструкція може бути реалізована у вигляді звичайних методів всередині типу, що додає мові певного об’єктно-орієнтованого вигляду. Не зважаючи на свою простоту методи типів все ж відрізняються від аналогічних в C#.

Розглянемо приклад:

```
type User(name: str, type: str)
{
   this FullName = name + " " + type
   this func GetName() => name
   static func Admin(name: str) => new User(name, "admin")
   static Empty = new User("", "")
   default = Empty
}
```

Не звичними у випадку TinyLang є використання ключових слів:

— this – визначає екземплярні поля та методи;

— static – визначає статичні поля та методи;

— default – визначає значення по замовчуванню для типу.

Цікавою особливістю TinyLang є використання оператору with. Оператор with дозволяє модифікувати екземпляр типу та повертати його копію з модифікованими полями не змінюючи початкового об’єкту.

Приклад:

```
type User(name: str, age: int)
func happyBirthday(user: User) => user with { age = age + 1}
vlad = new User("Vlad", 20)
vlad = happyBirthday(vlad)
```

Функція імпорту сторонніх бібліотек є однією з найважливіших при розробці мови програмування. У випадку TinyLang конструкція imports виконує ту ж саму функцію, що і аналогічна їй в C#. Також завдяки повній підтримці платформи .NET, сторонні бібліотеки не обов’язково повинні бути написані на мові програмування TinyLang та можуть бути написані на будь-якій .NET-сумісній мові.

Приклад використання конструкції:

```
imports Sytem.Collections
lst = new List<int>()
lst.AddRange(seq(1, 2, 3, 4, 5))
for(x in lst) print(x)
```

3.3 Порівняння з існуючими мовами програмування

Найбільш популярними мовами загального призначення на даний момент є: C#, Java, Python, C++ та JavaScript. Мова TinyLang володіє певними особливостями, які виділяють її серед цих мов. В рамках платформи .Net найбільш популярною мовою програмування є C#.

Порівняємо синтаксис TinyLang та основної мови платформи .NET – C# та розглянемо більш декларативні приклади обох мов програмування:

Таблиця 3.1 – Порівняння декларативного коду

Мова програмування	C#	TinyLang
Короткий опис роботи коду	Визначення власного типу прямокутника та обрахунок площі та периметра	

Код	using System;
	namespace TinyLangExamples
	{
	class Program
	{
	public class Rectangle
	{
	public int A { get; set; }
	public int B { get; set; }
	public Rectangle(int a, int b)
	{
	A = a;
	B = b;
	}
	}
	public static int Square(int a, int b) => a * b;
	public static int Perimeter(int a, int b) => (2 * a) + (2 * b);
	
	public static int
		type Rectangle(A: int, B: int)
		func Calculate(rect: Rectangle, formula: (int, int) -> int) =>
		formula(rect.A, rect.B)
		square = (a: int, b: int) => a * b
		perimeter = (a: int, b: int) =>
		(a * 2) + (b * 2)
		println(Calculate(new Rectangle(2, 4), square))
		println(Calculate(new Rectangle(2, 4), perimeter))

	Calculate(Rectangle rect, Func<int, int, int> formula) => formula(rect.A, rect.B);
	
	static void Main(string[] args)
	{
	Console.WriteLine(Calculate(new Rectangle(2, 4), Square));
	
	Console.WriteLine(Calculate(new Rectangle(2, 4), Perimeter));
	}
	}
	}	

Розглянемо максимально процедурні варіанти програм:

Таблиця 3.2 – Порівняння процедурного коду

Мова	C#	TinyLang
Короткий опис роботи коду	Виведення на екран матриці n*n та підрахунок загальної кількості елементів.	

Код	using System;
	namespace TinyLangExamples
	{
	class Program
	{
		func makeMatrix(size: int): int {
		for(i = 0 to size)
		{ for(j = 0 to size)

```
        static int MakeMatrix(int size)
        {
            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                {
                    var num = (i * size) + j;
                    if (num < 10)
                    {
            Console.Write("{0}   ", num);
                    }
                    else
                    {

                    Console.Write("{0} ", num);
                    }

                }
                Console.WriteLine();
            }

            return size * size;
        }
        static void Main(string[] args)
        {
Console.WriteLine(MakeMatrix(5));
        }
    }
}
```

```
 {
            num = (i *
size) + j
            if(num < 10)
printF("{0}   ",
num) else
            {
                printF("{0
} ",
                        num)
            }
        }
        println("")
    }
    return size * size
}
printF("Total: {0}",
makeMatrix(5))
```

Можна виділити основні моменти які відрізняють TinyLang від C#:

— менша кількість ключових слів;

— відсутність крапки з комою;

— відсутність модифікаторів доступу;

— відсутність методу main;

— менша кількість символів для описання аналогічних операторів;

— використання «record» - типів.

Остання ж особливість зараз є досить популярною в нових мовах програмування та поступово додається у вже існуючі. Так наприклад, судячи з висловлювань Меда Торгерсена – основного дизайнера мови C# з Microsoft, така можливість також має з’явитись у новому релізі мови C# 9.0 [15].

Розглянемо реалізацію цієї синтаксичної конструкції мовою F# та порівняємо її з аналогічною в TinyLang:

Таблиця 3.3 – Порівняння F# та TinyLang

Мова програмування	F#	TinyLang
Короткий опис роботи коду	Визначення власного типу	

Код	type Rectangle = {
	A: int
	B: int
	}
	
	[<EntryPoint>]
	let main argv =
	let rect = { A = 10; B = 20 }
	
	printfn "%s" (rect.ToString())
	0
		type Rectangle(A: int, B: int)
		rect = new Rectangle(10, 20)
		print(rect)

Вивід програми	{ A = 10 B = 20 }	Rectangle(A(10), B(20))

Можна помітити деяку схожість, як наприклад ключове слово «type» та позначення типу після ідентифікатору, але також є досить багато відмінностей: використання ключового слова «new» та в загальному менша кількість коду у випадку з TinyLang. Хоча мова програмування F# має також свої переваги реалізації визначення типів. Так, наприклад, при створенні екземпляру типу не обов'язково вказувати ім'я, тип буде визначений виключно по іменам полів. Обидві мови мають також автоматичну реалізацію методів приведення об'єктів в текстовий вигляд, що дозволяє покращити вивід в термінал роботи приведених прикладів програм.

Інші популярні мови програмування такі як Java, C++ чи Python не мають подібних можливостей і надають досить громіздкий створення типів.

3.4 Середовище розробки

Для можливості написання, редагування та запуску програм написаних на розробленій мові програмування було створено інтегроване середовище розробника (IDE – integrated development environment) з використанням мови програмування C# та платформи WPF, а також таких бібліотек як AvalonEdit та System.Reactive.

Основними можливостями якого стали:

— редагування файлів *.tl (описаних мовою TinyLang);
— підсвічення синтаксису;
— виявлення синтаксичних помилок на етапі редагування;
— вивід помилок при компіляції;
— запуск файлів *.tl;
— інтегрована консоль cmd;
— вивід згенерованого AST;
— імпорт та експорт AST у форматі JSON.

Створене середовище завдяки виділенню лише основного функціоналу є досить легковмісним та має порівняно високу швидкодію.

Розглянемо основні можливості середовища розробки в дії. Середовище розробки являється звичайним файлом для виконання формату .exe та може бути запущений на операційній системі Windows зі встановленим .Net Core версії 3.1 незалежно від розрядності системи. Для кращої взаємодії з середовищем розробника передбачається також розробка інсталятора, що забезпечить швидке встановлення всіх пакетів необхідних для розробки, включаючи: файли компілятора, середовища розробки, залежності у вигляді динамічних бібліотек, а також запис конфігурацій користувача та встановлення змінних середовища для виклику компілятора з командної стрічки. Проте на даний момент для запуску середовища розробки потрібно запустити файл «TinyLang.IDE.exe» і після цього середовище вже повністю готове до написання та запуску перших команд.

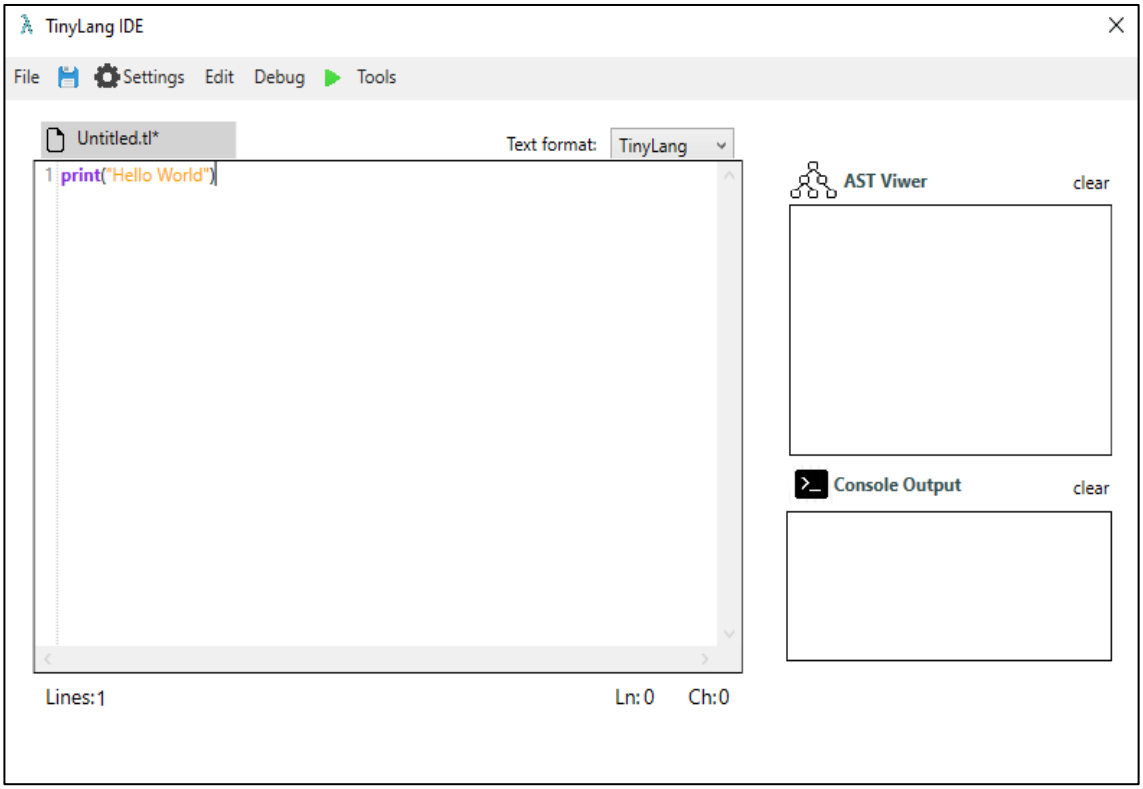

Рисунок 3.1 – Початкова форма середовища розробника

Одразу на початковій форма можна побачити 3 основних вікна, які використовуються для розробки. Перше вікно призначене для вводу коду та може приймати 2 основних формати:

— TinyLang – код синтаксису згенерованої мови;

— JSON – код AST мови TinyLang.

Середовище розробки має ряд команд швидкого доступу:

— Ctrl+Z – відміна останньої операції;

— Ctrl+Y – повернення до відміненої операції;

— Ctrl+S – збереження редагованого файлу, або ж збереження введеного коду в якості нового файлу;

— Ctrl+F5 – компіляція та запуск програми;

— Ctrl+O – відкриття нового файлу.

При введенні та запуску коду програми, створюється Абстрактне синтаксичне дерево, що представляє всі введені операції, а також, якщо були викликані функції виводу в консоль, ця інформація додається до вікна консолі, або ж відкривається нове вікно терміналу Windows в залежності від вибраних налаштувань.

Після запуску команди Debug->Run (Ctrl+F5) можна спостерігати створення абстрактного синтаксичного дерева у вікні «AST Viewer» та виводу в команду стрічку вікна «Console Output»:

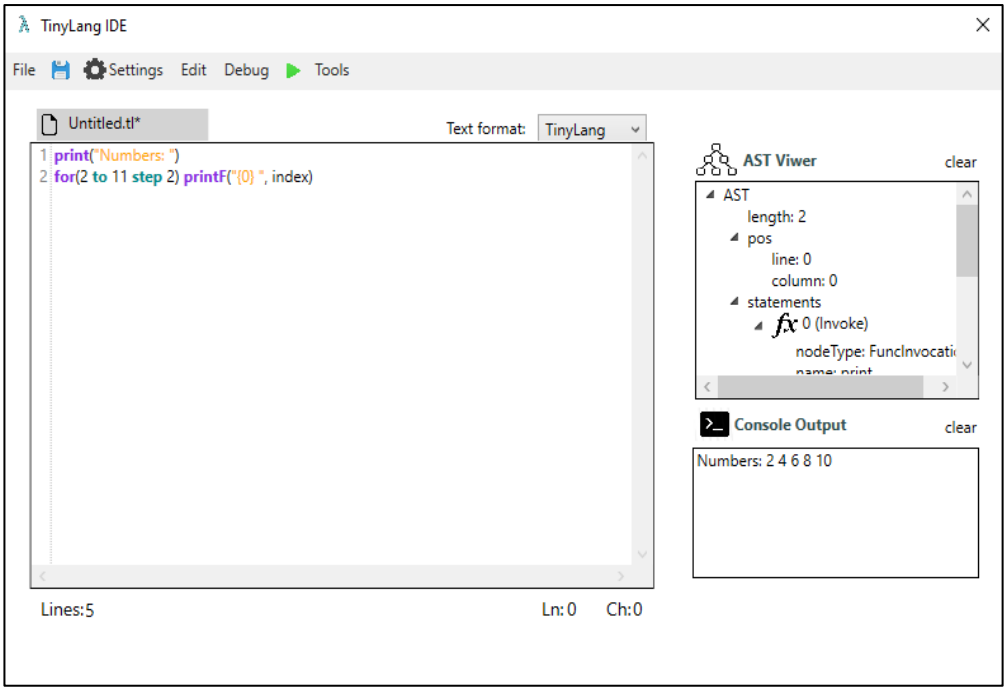

Рисунок 3.2 – Вивід результатів виконання програми

При введенні тексту програми, який містить синтаксичну помилку, позиція в тексті де починається помилка підкреслюється червоною лінією, а при компіляції виводиться текст помилки:

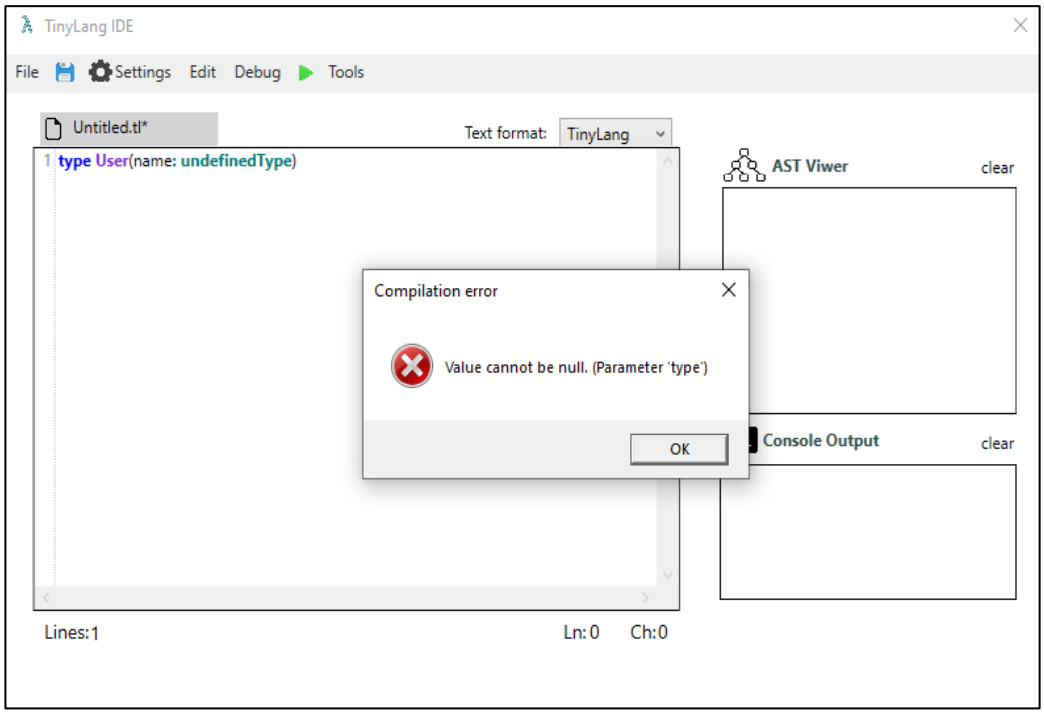

Рисунок 3.3 – Обробка помилок компіляції

В коді, приведеному на рисунку 21 використовується невизначений тип – «undefinedType», тому при компіляції коду виникла помилка про неіснуючий тип. Для обробки помилок, та інших важливих частин середовище розробки є окрема форма з налаштуваннями. На даний момент є 2 основні вкладки: «General Settings» та «Exceptions». Перша містить основні налаштування середовища, такі як:

— тип виводу результатів програми;
— тема;
— вивід AST;
— режим дебагу.

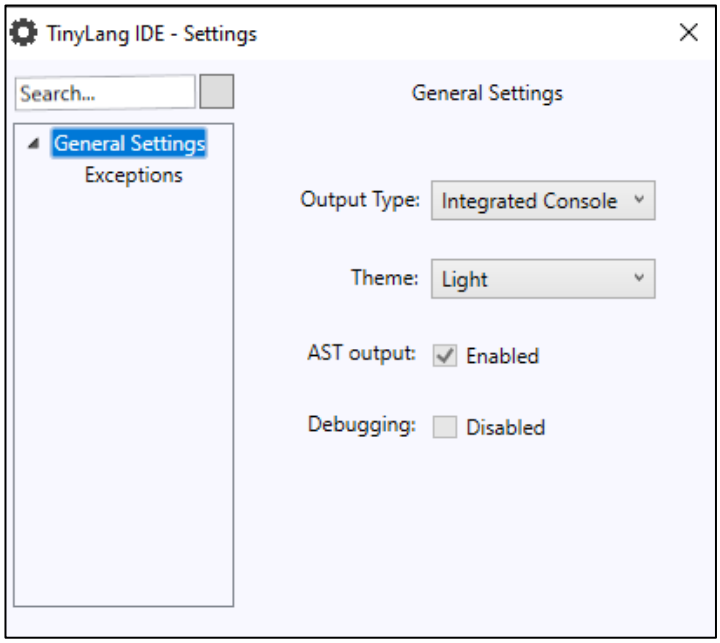

Рисунок 3.4 – Форма основних налаштувань

Форма «Exception» – містить опції обробки помилок:

— помилки при компіляції;

— «розумні» підказки.

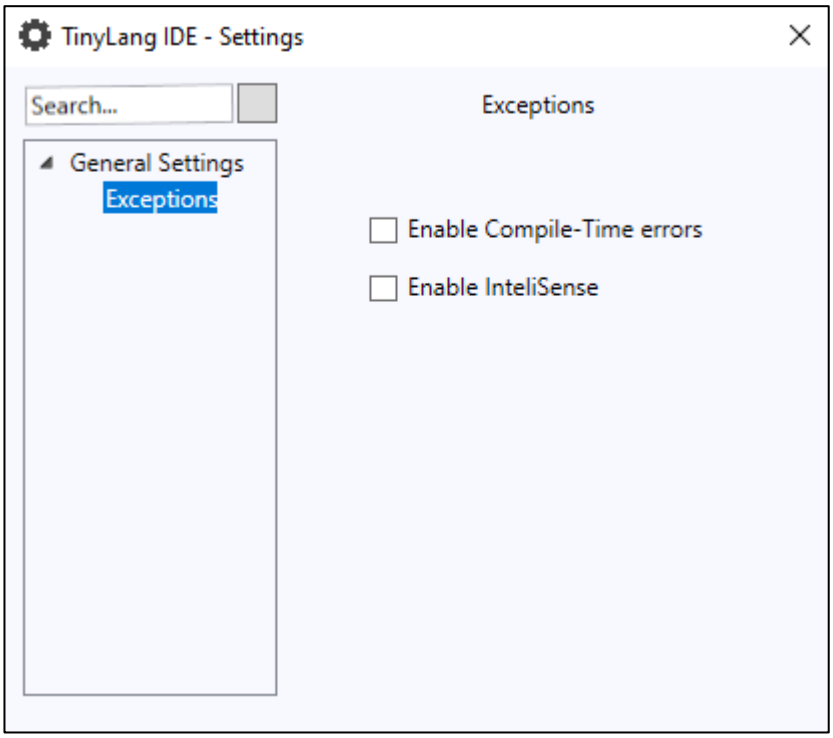

Рисунок 3.5 – Форма налаштувань обробки помилок

На початковій стадії розробки та відлагодження мови, дуже важливими є функції роботи з абстрактними синтаксичними деревами.

При ввімкненій опції виводу AST, його можливо експортувати або ж імпортувати у форматі JSON. Для цього на основній формі є кнопка «export», яка знаходить над вікном виводу AST. Після її натиснення з'являється вікно збереження файлу – рисунок 3.6.

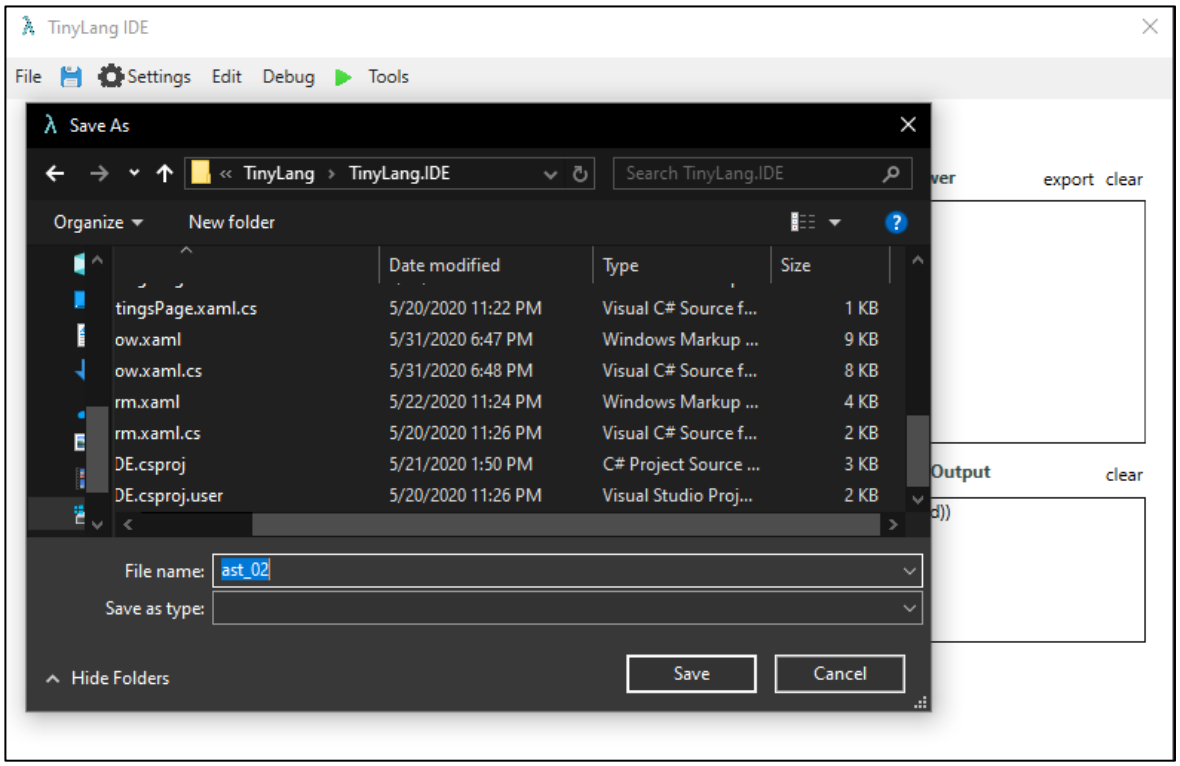

Рисунок 3.6 – Експорт AST

По замовчуванню, для збереження, відкривається директорія з якої було запущено середовище розробника, проте така поведінка може бути змінена в налаштуваннях користувача.

Після збереження файлу, його можна буде імпортувати за допомогою окремої функції, яка може бути викликана з меню відкриття файлу, в якому пропонується вибір з 2-х можливих розширення для відкриття: «*.json» та «*.tl».

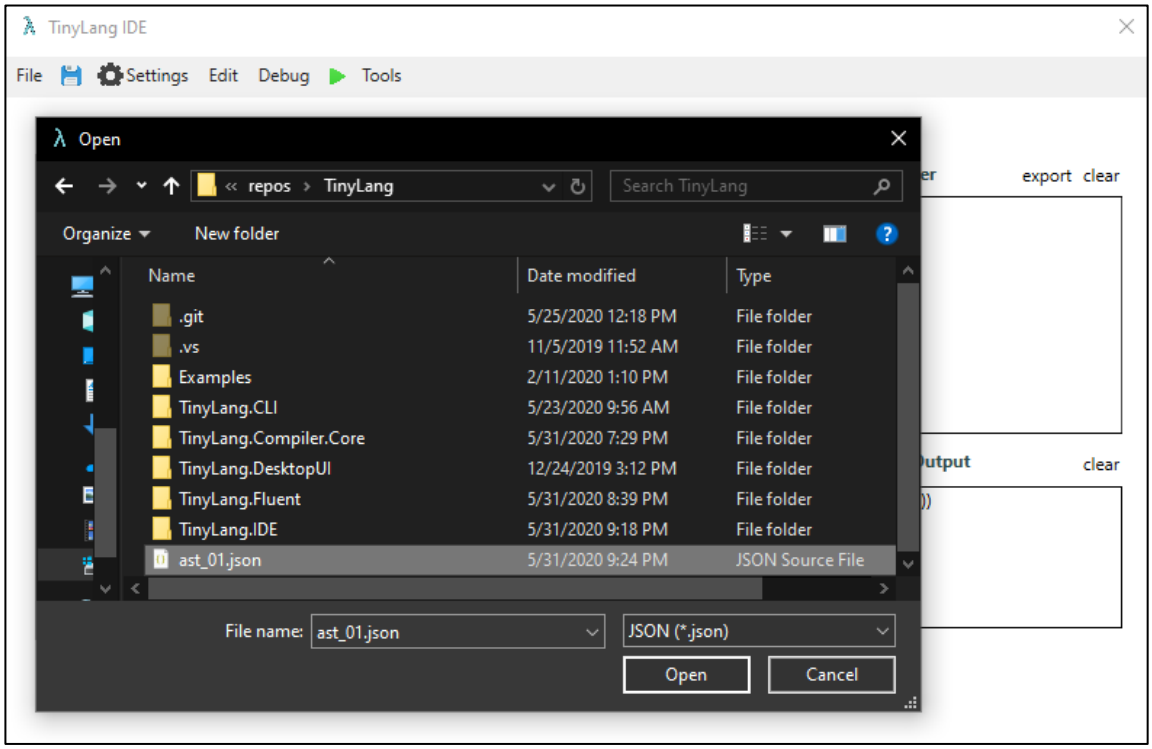

Рисунок 3.7 – Імпорт AST

Після відкриття файлу в одному з доступних форматів, його вміст та назва поміщається до вікна редагування. При цьому формат тексту також відображається на формі, хоча і може бути змінений незалежно від типу відкритого файлу. Від зміни формату тексту залежить підсвічування синтаксису а також виділення помилок. Текст файлу також може бути змінений або ж запущений на виконання незалежно від формату. Так можна писати код мовою TinyLang, потім зберігати у форматі JSON, імпортувати назад та мати можливість редагувати вже згенероване синтаксичним аналізатором абстрактне синтаксичне дерево. Ця функція відкриває можливість написання модулів генерацій коду для ще не реалізованих синтаксичних конструкцій та перевіряти їх роботу використовуючи AST.

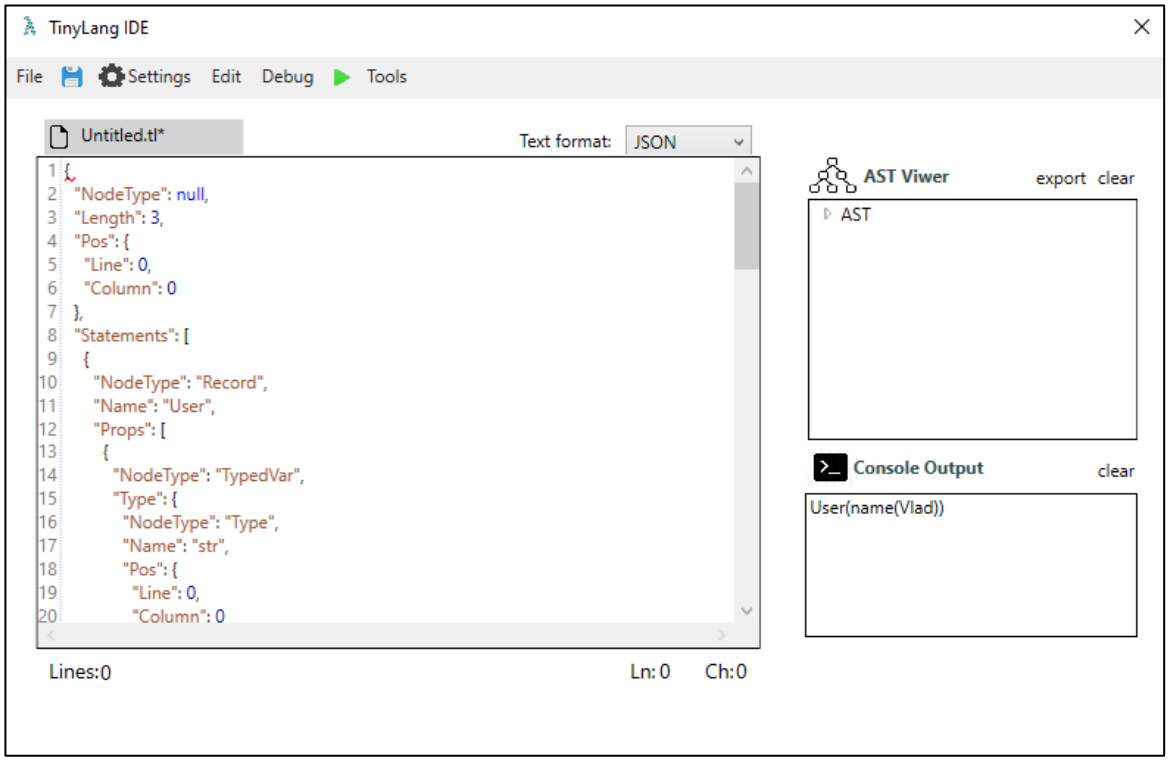

Рисунок 3.8 – Запуск AST

Створена мова програмування а також середовище розробника було протестоване на прикладних задачах та у більшості випадків дозволяло значно зменшити кількість коду та час на написання. Обробка помилок, а також вивід абстрактного синтаксичного дерева дозволяє ефективніше писати код. Створене в даному розділі інтегроване середовище розробника може забезпечити повноцінну розробку застосунків, а також бути корисним при подальшій розробці компілятору так як надає функціонал виводу та модифікації абстрактного синтаксичного дерева. Хоча, середовище, поки що, надає лише базовий функціонал, проте є повністю готовим до апробації компілятору та мови програмування на прикладних задачах.

4 ВИРІШЕННЯ ПРИКЛАДНИХ ЗАДАЧ МОВОЮ TINYLANG

Створена мова програмування є мовою загального призначення, проте на даному етапі через відсутність прикладних бібліотек та підтримки з боку синтаксису певних конструкцій має хоч і обмежену, проте досить обширну область застосування. Мова програмування TinyLang на початку свого формування має 3 основні напрямки застосування:

— IoT;

— скрипти автоматизації;

— хмарні обчислення.

У більшості випадків задач в наведених областях мова може створити достатню конкуренцію вже існуючим. В деяких випадках завдяки спрощеному синтаксису, в інших же, завдяки більшим можливостям. Основними акцентами при вирішенні прикладних задач є функціональний підхід та декларативність. Завдяки можливості використання функцій вищого порядку та спрощеному синтаксису визначення типів, можна писати код досить високої якості. Також, в майбутньому, мова має всі шанси до застосування в розробці мікросервісів для великих, корпоративних WEB застосунків, надаючи всі можливості таких мов як C# або Java, проте пропонуючи значно спрощений синтаксис та декларативність. Саме ця особливість може забезпечити швидке поширення мови серед початківців та в загальному підвищити популярність мови.

4.1 Застосування в IoT задачах

IoT, або ж Інтернет Речей – набираюча популярність галузь програмної та системної інженерії. Основна її концепція – створення комунікації між певними, окремими пристроями, для вирішення конкретних задач. Такими пристроями можуть бути: сенсори, датчики, давачі та навіть смартфони. Для того щоб створити таку комунікацію, крім апаратного, потрібне також програмне забезпечення. Найбільш популярними мовами в галузі є: C, C++, Java, Python, Rust, Go та JavaScript. У більшості цих мов, зважаючи на історію їх

формування є певний ряд недоліків: будь-то занадто громіздкий синтаксис, або ж динамічна типізація. Хоча й більшість цих недоліків поступово вирішуються творцями мов, проте початкова конструкція мови та компілятора не дозволяє робити значних змін в синтаксисі. Для більш декларативного опису роботи мережі IoT пристроїв можна застосувати такі конструкції мови TinyLang як: спрощене визначення власних типів, спрощений визначення функцій, спрощені конструкції циклів, відсутність точки входу в програму.

Наприклад, визначення типів та функцій для системи регулювання кліматом в кімнаті може бути представлене наступним чином:

```
type MoistureInfo(Percent: int)
type TemperatureInfo(Celsius: int)
type ClimatInfo(Moisture: MoistureInfo, Temperature: TemperatureInfo)
func CallClimatSensors(moisture: int, temp: int)
{
   climat = new ClimatInfo(new MoistureInfo(moisture),
      new TemperatureInfo(temp))
   return http.Post("localhost/climatcontrol", climat).Response.StatusCode
      == HttpStatusCode.Ok ? climat : null
}
func CallClimatControl(sensors: ClimatInfo)
   => sensors != null ? "Temperature and moisture regulated!" : "error"
print(CallClimatControl(CallClimatSensors(55, 30)))
```

Спочатку визначаються всі типи для роботи програми, потім визначаються основні функції. Остання операція викликає визначені функції та приводить в роботу програму. В даному випадку конкретна реалізація функцій для простити упускається та найважливішу роль відіграє саме вид запису. Аналогічні ж конструкції на C++, Python або Java були б більш

громіздкими.

В майбутньому при повній реалізації нативної підтримки бібліотек для мережевої комунікації збоку синтаксису, а також завершення генерації ADT,

реалізація методів виглядала досить декларативно. Після повної реалізації всіх синтаксичних конструкцій мова б могла конкурувати по декларативністю з більшістю представлених мов програмування в галуззі.

4.2 Застосування в якості скриптів автоматизації

Важливу роль в роботі з будь-якою операційною системою при вирішенні прикладних задач – є автоматизація певних рутинних задач за допомогою скриптів. Наприклад вбудованими мовами для операційної системи Windows є Batch та PowerShell. Останній же досить добре себе зарекомендував в якості скриптової мови, проте все ж передбачає вивчення, так як має хоч і досить простий, але все ж унікальний синтаксис. Тому при розробці основного додатка на певній мові програмування, було б досить зручно писати скрипти автоматизації на ній же. Тому мова програмування TinyLang досить добре вписується в таку концепцію. Наприклад скрипт для перейменування файлів міг би виглядати так:

```
VERSION = "1.2.3.0"
for(file in directory("/foo/bar")
{
   if(contains(file.NameWithoutExtension, "foobar"))
   {
      rename(file, file.NameWithoutExtension + "_" + VERSION)
   }
}
```

Особливо декларативно в даному випадку виглядає запис “for(file in directory)”, який упускає деталі роботи з індексами та колекціями та виглядає досить очевидно. Такий підхід формує основне уявлення про концепцію мови. З появленням нових подібних конструкцій, мова буде виглядати все більше схожу на людську. Також завдяки можливостям компілятору, більшість фігурних дужок модна упустити і код буде виглядати наступним чином:

```
VERSION = "1.2.3.0"
for(file in directory("/foo/bar")
if(contains(file.NameWithoutExtension, "foobar"))
rename(file, file.NameWithoutExtension + "_" + VERSION)
```

Такий вид запису навіть не потребує додаткової табуляції та може працювати з довільною кількістю пробілів, а кінець блоку визначається з контексту наступного виразу. Це досить сильно наближує мови до PowerShell, але може бути застосоване не лише в скриптах, а й в повноцінних додатках.

Така відсутність додаткових обмежень теж має досить велику вагу при вивченні нової мови.

4.3 Застосування в хмарних обчисленнях

Ще одна, можлива область застосування спроєктованої мови програмування – хмарні обчислення. Все більше розробників веб ресурсів переносять свої додатки в хмарні середовища такі як: Microsoft Azure, Amazon AWS, Google Cloud та багато інших. Однією з найцікавіших можливостей хмарних середовищ є PaaS (Platform as a Service) та SaaS (Software as a Service) рішення, які дозволять досить абстрактно описувати логіку роботи з програмою, а всі інфраструктурні та операційні особливості вирішуються автоматично засобами хмарного провайдера. Серед таких рішень особливе місце займають – хмарні функції, наприклад Azure Functions, або Amazon AWS.

Розглянемо для прикладу Azure functions. Функції Azure дозволяють запускати невеликі фрагменти коду (так звані «функції»), не турбуючись про інфраструктуру додатків. За допомогою функцій Azure хмарна інфраструктура надає всі сучасні сервери, необхідні для того, щоб програма працювала та масштабувалася автоматично. Функція «спрацьовує» на конкретний тип події. Підтримувані тригери включають реагування на зміни в даних, відповідь на повідомлення, виконання за графіком або в результаті запиту HTTP.

Для коректної роботи функцій на специфічній мові програмування потрібна повна підтримка компілятора та аналізатора цієї мови від провайдера

хмарних сервісів. Так наприклад Azure Functions підтримують лише такі мови як: C#, Java, JavaScript, Python та PowerShell. Але цей перелік поступово поповнюється новими мовами та не виключено, що TinyLang може стати наступною.

Не зважаючи на відсутність підтримки мови в середовищі Azure, реалізація такої функції в TinyLang могла б виглядати наступним чином:

```
<HttpVerb.Get("/hello")>
func ProcessRequest(request: HttpRequest): IActionResult
{
   if (request.QueryString == "hello")
   {
      return Ok("Hello World!")
   }
   else
   {
      return BadRequest()
   }
}
```

Така реалізація потребує також імплементації в мову програмування атрибутів, але дозволяє досить швидко та декларативно описувати Http запити хмарної функції.

Хоча імплементація більшості тригерів функцій Azure є досить складною, HTTP тригери не здаються такими та можуть бути додані досить просто. Завдяки також можливості використання в якості скриптів автоматизації, мова може використовуватися в CI/CD процесах.

Проаналізовані в даному розділі приклади хоч і не домонструють всіх синтаксичниї можливостей мови, проте показують її переваги та декларативність.

Використовуючись вже в 3-х окремих галуззях TinyLang є перспективним інструментом в арсеналі системного інженера так як, забезпечує швидку розробку застосунків. В майбутньому, зі збільшенням можливостей, мова

може розширити свою цільову аудиторію та стати основною при розробці повноцінних додатків. Забезпечуючи спрощений синтаксис та підвищену декларативність, мова може вигідно вирізнятись на фоні інших при вибори мови для розробки програмного забезпечення для IoT мережі, мікросервісів та хмарних функцій, або ж звичайних скриптів.

ВИСНОВКИ

В ході дипломного проєктування було розглянуто фундаментальні особливості побудови компіляторів а також дизайну мов програмування. Проаналізувавши існуючі мови програмування та виявшивши їх недоліки, було запропоновано створити власну мову прорграмування.

В результаті дипломної роботи виконано:

а) Розробку синтаксису нової мови програмування TinyLang, яка поєднує у собі скриптовий, об’єктно-орієнтований та функціональний підходи.

б) Розробку компілятора мови програмування TinyLang з використанням комбінаторів парсерів.

в) Створення інтегрованого середовища розробки, яке надає функціонал компіляції та виконання програм мовою TinyLang.

г) Апробацію розробленої мови програмування для вирішення прикладних задач.

Створена мова має наступні переваги:

— Спрощений синтаксис

— Підтримка стандартної бібліотеки .Net

— Сумісність з іншими мовами .Net

— Кроссплатформенність

— Можливість використання компілятора в якості бібліотеки

Розроблене інтегроване середовище розробника надає повну підтримку синтаксису з виявленням помилок на етапах редагування коду та компіляції.

Мова програмування TinyLang завдяки своїй простоті та потужності може знайти своє застування в багатьох галузях, проте на початковому етапі основним є: програмування IoT мереж, системне програмування та хмарні обчислення.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Сергій Лідін. “Expert .NET 2.0 IL Assembler”. 2006 р. 530 ст.;

2) Дік Груне. “Modern Compiler Design”. 2012 р. 829 ст.;

3) Едвард Дж. Нілгес. “Build Your Own .NET Language and Compiler”. 2004 р. 408 ст.;

4) Дік Груне. “Parsing Techniques: A Practical Guide”. 2010 р. 688 ст.;

5) Альфред Ахо. “Compilers: Principles, Techniques and Tools”. 1986 р. 1184 ст.;

6) Михаель Скот. “Programming Language Pragmatics”. 2015 р. 992 ст.;

7) Monadic Parser Combinators In C#. Використання монадних парсер комбінаторів в C#. URL: https://tyrrrz.me/blog/monadic-parser-combinators;

8) Efficient parsing with parser combinators. Software Composition Group, University of Bern, Switzerland 2015. URL: https://www.sciencedirect.com/science/article/abs/pii/S0167642317302654;

9) Monadic Parser Combinators - Graham Hutton University of Nottingham. URL: https://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf ;

10) Lexical analysis with parser combinators. URL: https://ro-che.info/articles/2015-01-02-lexical-analysis – Назва з екрану;

11) Теленик С. Ф. Спеціальні розділи математики. Дискретна математика. Частина 2. URL: https://ela.kpi.ua/handle/123456789/12580;

12) Introduction of Compiler Design. Вступ до проєктування компілятоів. URL: https://www.geeksforgeeks.org/introduction-of-compiler-design/ ;

13) The New Trends in Compiler Analysis and Optimizations. International Journal of Emerging Trends & Technology in Computer Science. 2017 р. URL: https://www.researchgate.net/publication/316791432_The_New_Trends_in_Compiler_Analysis_and_Optimizations;

14) Compiler testing: a systematic literature analysis. Стаття журналу «Frontiers of Computer Science». 2019 р. URL: https://link.springer.com/article/10.1007/s11704-019-8231-0;

15) Медс Торгерсен. Welcome to C# 9.0. Майбутні нові можливості C# 9.0. 2020р. URL: https://devblogs.microsoft.com/dotnet/welcome-to-c-9-0/;

16) Михаель Вольф. Compiling history to understand the future. 2018 р. URL: https://www.nextplatform.com/2018/11/02/compiling-history-to-understand-the-future/;

Додаток А

EBNF специфікація мови

Таблиця А.1

Назва виразу	Опис	Правило виводу
identifier	Ідентифікатор	{ "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0" | "_"}
space	Пробіл	" "
string	Текстове значення	""", {identifier | space}, """
integer	Цілочисельне значення	{0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
bool	Булеве значення	"true" | "false"
variable	Ім’я змінної	identifier
typeDeclaration	Визначення типу	":", {space}, type
type	Ім’я типу	(identifier | funcType)
typedArgument	Типізований аргумент	identifier, typeDeclaration
typeDeclaration	Визначення власного типу	"type", {space}, identifier, "(" { (typedArgument, [“,”]) }, ")"
valueExpression	Вираз, що повертає значення	string | integer | bool | variable | funcCall | newType | ternaryOp | mathExpression | boolExpression | stringConcat | lambda
expression	Вираз	valueExpression | ifElifElse | for | forEach | while | doWhile
scope	Блок виразів	"{", { expression | space}, "}"
return	Повернення результату з функції	"return", {space}, valueExpression
funcScope	Блок функції	"{", { expression | space}, [return], "}"
variableDeclaration	Визначення змінної	identifier, {space}, "=", {space}, valueExpression
funcArgs	Аргументи функції	"(" { (typedArgument, [“,”]) } ")", [typeDeclaration]
funcDeclaration	Визначення нової функції	"func", {space}, identifier, funcArgs, funcScope

Продовження таблиці А.1

lambdaBody	Тіло лямбда функції	"=>", {space}, valueExpression
Lambda	Лямбда функції	funcArgs, lambdaBody
funcType	Тип функції	"(", {(type, [","])}, ")", "->", type
funcCall	Виклик функції	identifier, ("(",{(valueExpression, [","])}, ")"
newType	Створення типу	"new", identifier, " (", {valueExpression | [","]}, ")"
mathExpression	Математичний вираз	valueExpression, {space}, ("-" | "+" | "*" | "/"), {space}, valueExpression
boolExpression	Булевий вираз	(["!"], valueExpression), {space}, ("&&" | "||"), {space} (["!"], valueExpression)
equality	Порівняння на рівність	valueExpression, {space}, ("!=","=="), {space}, valueExpression
comprassion	Порівняння	valueExpression, {space}, (">","<"), {space}, valueExpression
ternaryExpression	Тернарний оператор	valueExpression, {space}, "?", {space}, valueExpression, {space}, ":", {space}, valueExpression
stringConcat	Конктенація стрічок	valueExpression, {space}, "+", {space}, valueExpression
elif	Гілка “if else” оператору розгалуження	"elif", {space}, "(", {space}, valueExpression, {space}, ")", scope
else	Гілка “else”	"else", {space}, scope
ifElifElse	Оператор розгалуження	"if", {space}, "(", {space}, valueExpression, {space}, ")", scope, {space}, [elif], [else]
for	Цикл “For”	"for", {space}, "(", {space}, (integer | variableDeclaration), {space}, ")", {space}, (scope | expression)
forEach	Цикл “ForEach”	"for", {space}, "(", {space}, variable, {space}, "in", {space}, variable, ")", {space}, (scope | expression)

Додаток Б

Вихіний код основних класів компілятора

```
public class Position
{
    public int Line { get; set; }
    public int Column { get; set; }

    public Position(int line, int column)
    {
        Line = line;
        Column = column;
    }

    public Position(Pos pos) : this(pos.Line, pos.Column) { }

    public override string ToString()
    {
        return $"{{ line: {Line}, column: {Column} }}";
    }
}

public abstract class Expr
{
    [JsonProperty(Order = -2)]
    public virtual string NodeType => GetType().Name.Replace(nameof(Expr),
string.Empty);

    public Position Pos { get; set; } = new Position(0, 0);

    public Expr WithPosition(int line, int column)
    {
        Pos = new Position(line, column);
        return this;
    }

    public Expr WithPosition(Pos p) => WithPosition(p.Line, p.Column);

    public static Expr Bool(bool value) => new BoolExpr(value);
    public static Expr Int(int value) => new IntExpr(value);
    public static Expr Str(string value) => new StrExpr(value);
    public static Expr Add(Expr left, Expr right) => new AddExpr(left, right);
    public static Expr Div(Expr left, Expr right) => new DivExpr(left, right);
    public static Expr Subtr(Expr left, Expr right) => new SubtrExpr(left, left);
```

```
        public static Expr Mul(Expr left, Expr right) => new MulExpr(left, right);
        public static Expr And(Expr left, Expr right) => new AndExpr(left, right);
        public static Expr Or(Expr left, Expr right) => new OrExpr(left, right);
        public static Expr Not(Expr value) => new NotExpr(value);
        public static Expr Var(string name) => new VarExpr(name);
        public static Expr Assign(Expr assignment, Expr value) => new
AssignExpr(assignment, value);
        public static Expr Eq(Expr left, Expr right) => new EqExpr(left, right);
        public static Expr NotEq(Expr left, Expr right) => new NotEqExpr(left,
right);
        public static Expr Less(Expr left, Expr right) => new LessExpr(left, right);
        public static Expr LessOrEq(Expr left, Expr right) => new
LessOrEqExpr(left, right);
        public static Expr More(Expr left, Expr right) => new MoreExpr(left, right);
        public static Expr MoreOrEq(Expr left, Expr right) => new
MoreOrEqExpr(left,



right);
        public static Expr If(Expr condition, Expr result) => new
TernaryIfExpr(condition, result);
        public static Expr Choose(Expr left, Expr right) => new ChooseExpr(left,
right);
    }
[JsonObject]
    public sealed class AST : Expr, IEnumerable<Expr>
    {
        [JsonProperty(Order=2)]
        public Expr[] Statements { get; }

        public override string NodeType => null;

        public int Length => Statements.Length;

        public AST(params Expr[] args) => Statements = args;

        public AST(IEnumerable<Expr> exprs) => Statements = exprs.ToArray();

        public override string ToString()
        {
            return JsonConvert.SerializeObject(this, settings: new
JsonSerializerSettings
            {
                Formatting = Formatting.Indented,
                StringEscapeHandling = StringEscapeHandling.EscapeNonAscii,
                ContractResolver = new CamelCasePropertyNamesContractResolver(),
```

```
            NullValueHandling = NullValueHandling.Ignore
        });
    }

    public IEnumerator<Expr> GetEnumerator() =>
Statements.AsEnumerable().GetEnumerator();

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    public static implicit operator AST(Expr[] exprs) => new AST(exprs);
  }
public class Scope : Expr
  {
    public List<Expr> Statements { get; }

    public Scope(List<Expr> statements)
    {
      Statements = statements;
    }

    public Scope(params Expr[] statements)
    {
      Statements = statements.ToList();
    }

    public Scope()
    {
      Statements = new List<Expr>();
    }

    public override string ToString()
    {
      return $"Scope({string.Join(", ", Statements)})";
    }



    public class EndOfScope : Expr
    {

    }

    public class StartOfScope : Expr
    {

    }
```

```
    }

    public class ScopedExpr : Expr
    {
        public Scope Scope { get; set; }
    }
public interface IASTBuilder
    {
        IASTBuilder Empty();

        IASTBuilder FromStr(string str);

        IASTBuilder AddStatement(Expr e);

        AST Build();
    }

    public class ASTBuilder : IASTBuilder
    {
        private Parser<Expr> _parser;
        private AST _ast;

        public ASTBuilder(IParserBuilder<Expr> exprParserBuilder,
ITokenizer<Expr> exprTokenizer)
        {
            _parser = exprTokenizer.Tokenize(exprParserBuilder);
        }

        public AST Build() => _ast;

        public IASTBuilder FromStr(string str)
        {
            var parser =
                    from s in spaces from x in many1(_parser)
                    select x.AsEnumerable();

            _ast = new AST(ParseWithExceptionThrow(parser, str));

            return this;
        }

        public IASTBuilder AddStatement(Expr e)
        {
            _ast = _ast.Append(e).ToArray();

            return this;
```

```
        }

        private T ParseWithExceptionThrow<T>(Parser<T> parser, string line)
        {
            var parsed = parse(parser, line);
            var error = parsed.Reply.Error;


            if (error.Tag == ParserErrorTag.Message
                || (error.Msg != "end of stream" && error.Tag ==
ParserErrorTag.Expect))
            {
                throw new PositionedException(new Position(error.Pos), "parser error");
            }

            return parsed.IsFaulted ? throw new PositionedException(new
Position(error.Pos), "parser error")
                : parsed.Reply.Result;
        }

        public IASTBuilder Empty()
        {
            _ast = new AST();
            return this;
        }
    }
public interface ITokenizer<T>
    {
        Parser<T> Tokenize(IParserBuilder<T> expressionParserBuilder);
    }

    public class ExprTokenizer : ITokenizer<Expr>
    {
        public Parser<Expr> Tokenize(IParserBuilder<Expr>
expressionParserBuilder)
        {
            var exprParser = expressionParserBuilder
                .WithBinaryOperation("*", Assoc.Left, Expr.Mul, 4)
                .WithBinaryOperation("+", Assoc.Left, Expr.Add, 3)
                .WithBinaryOperation("-", Assoc.Left, Expr.Subtr, 3)
                .WithBinaryOperation("/", Assoc.Left, Expr.Div, 4)
                .WithBinaryOperation("&&", Assoc.Left, Expr.And, 3)
                .WithBinaryOperation("=", Assoc.Left, Expr.Assign, 0)
                .WithBinaryOperation("==", Assoc.Left, Expr.Eq, 3)
                .WithBinaryOperation("!=", Assoc.Left, Expr.NotEq, 3)
                .WithBinaryOperation("<", Assoc.Left, Expr.Less, 3)
```

```
            .WithBinaryOperation("<=", Assoc.Left, Expr.LessOrEq, 3)
            .WithBinaryOperation(">", Assoc.Left, Expr.More, 3)
            .WithBinaryOperation(">=", Assoc.Left, Expr.MoreOrEq, 3)
            .WithBinaryOperation("?", Assoc.Left, Expr.If, 1)
            .WithBinaryOperation(":", Assoc.Left, Expr.Choose, 2)
            .WithUnaryOperation("!", UnaryOperationType.Prefix, Expr.Not, 2)
            .WithBinaryOperation("||", Assoc.Left, Expr.Or, 3)
            .Build();

        //exprParser = either(attempt(FuncInvocation(lazyp(() => exprParser))),
exprParser);

        var parser = Compose(exprParser, IfElse, For, While, DoWhile,
FuncDefinition);

        parser = Add(parser, Records);

        return parser;
    }

    private Parser<Expr> Compose(Parser<Expr> parser, params
Func<Parser<Expr>, Parser<Expr>>[] funcs)
    {
        var parsers = funcs.Select(f => f(attempt(lazyp(() => parser)))).ToArray();
        parser = either(attempt(choice(parsers)), parser);


        return parser;
    }

    private Parser<Expr> Add(Parser<Expr> parser, Func<Parser<Expr>> f)
    {
        return either(attempt(parser), f());
    }
  }
public static class TinyLanguage
  {

    public static GenLanguageDef LanguageDef { get; }

    public static GenTokenParser TokenParser { get; }

    public static Parser<Expr> IntParser { get; }

    public static Parser<Expr> BoolParser { get; }
```

```
        public static Parser<Expr> StrParser { get; }

        public static Parser<Expr> VarParser { get; }

        public static Parser<Expr> UntypedVarParser { get; set; }

        public static Parser<TypeExpr> TypeAssignParser { get; }

        public static Parser<Expr> TypedVarParser { get; }

        public static Parser<Expr> ExprValueParser { get; }

        public static Parser<string> IdentifierParser { get; }

        public static Parser<string> StrValue(string value) =>
            from str in asString(many1(letter))
            where str == value
            select str;

        public static Parser<Expr> GetExprValueParser(Parser<Expr> exprParser)
        {
            var parser = choice(
                attempt(RecordParsers.PropGetter(exprParser)),
                attempt(FuncInvocation(exprParser)),
                attempt(Lambda(exprParser)),
                attempt(RecordParsers.RecordCreation(exprParser)),
                attempt(BoolParser),
                attempt(IntParser),
                attempt(StrParser),
                VarParser);

            return from p in getPos
                   from e in parser
                   select e.WithPosition(p.Line, p.Column);
        }

        public static Parser<Expr> Scope(Parser<Expr> parser) =>
            from exprSet in TokenParser.Braces(many(parser))
            select new Scope(exprSet.ToList()) as Expr;

        public static Parser<Expr> ScopeOrSingle(Parser<Expr> parser) =>
    either(Scope(parser), Single(parser));
```

```
    public static Parser<Expr> Single(Parser<Expr> parser) => from s in
optional(space)
                                                    from expr in parser
                                                    select new Scope(expr) as Expr;

    static TinyLanguage()
    {
        LanguageDef = Language.JavaStyle.With(ReservedOpNames: new
Lst<string>(ReservedNames.All));

        TokenParser = makeTokenParser(LanguageDef);

        IntParser = from n in TokenParser.Natural
                    select Int(n);

        BoolParser = from w in asString(from w in many1(letter) from sc in
many(symbolchar) from sp in spaces select w)
                     where string.Equals(w, "true",
StringComparison.OrdinalIgnoreCase)
                           || string.Equals(w, "false",
StringComparison.OrdinalIgnoreCase)
                     select Bool(bool.Parse(w));

        StrParser = from p in getPos from str in TokenParser.StringLiteral select
Str(str).WithPosition(p);

        IdentifierParser = from w in asString(from word in many1(letter)
                                              from sp in spaces
                                              select word)
                           where !LanguageDef.ReservedOpNames.Contains(w)
                           select w;

        UntypedVarParser = from i in IdentifierParser
                           select new GeneralOperations.VarExpr(i) as Expr;

        var typeParser = from s in TokenParser.Identifier
                         select new TypeExpr(s);

        TypeAssignParser = from c in TokenParser.Colon
                           from t in either(attempt(FuncType()), typeParser)
                           select t;

        Func<GeneralOperations.VarExpr, Option<TypeExpr>, Expr> defineVar =
(v, t) =>
            t.Match<Expr>(some => new TypedVar(v, some), () => v);
```

```
            VarParser = from v in UntypedVarParser
                        // from t in optional(TypeAssignParser)
                        // select defineVar(v as GeneralOperations.VarExpr, t);
                    from p in getPos
                    select v.WithPosition(p);

            TypedVarParser = from v in UntypedVarParser
                        from t in TypeAssignParser
                        from p in getPos
                        select defineVar(v as GeneralOperations.VarExpr,
t).WithPosition(p);




            ExprValueParser = choice(attempt(BoolParser),
                attempt(IntParser), attempt(StrParser), VarParser);
        }
    }

public interface ICodeGenerator
    {
        CodeGenerationState Generate(Expr expression, CodeGenerationState state);
    }

    public abstract class CodeGenerator<TExpr> : ICodeGenerator where TExpr :
Expr
    {
        public readonly ICodeGeneratorsFactory Factory;
        protected CodeGenerator(ICodeGeneratorsFactory factory)
        {
            Factory = factory;
        }

        public CodeGenerationState Generate(Expr expression, CodeGenerationState
state)
        {
            return GenerateInternal(Typed(expression), state);
        }

        protected virtual TExpr Typed(Expr expr) => expr is TExpr typed ? typed
                : throw new ExprTypeMismatchException(typeof(TExpr),
expr.GetType(), expr.Pos);
```

```
        protected internal abstract CodeGenerationState GenerateInternal(TExpr
expression, CodeGenerationState state);

        protected TypedLoader ValueLoader(Expr expr, ILGenerator ilGenerator,
CodeGenerationState state)
            => TypedLoader.FromValue(expr, ilGenerator, state, Factory);

        protected void LoadScope(Scope scope, CodeGenerationState state)
        {
            foreach (var s in scope.Statements)
            {
                Factory.GeneratorFor(s.GetType()).Generate(s, state);
            }
        }


        protected TypedLoader RecordLoader
          (RecordCreationExpr expr, ILGenerator ilGenerator, CodeGenerationState
state)
            => TypedLoader.FromRecordCreation(expr, ilGenerator, state, Factory);
    }
public interface ICodeGeneratorsFactory
    {
        ICodeGenerator GeneratorFor(Type type, Type originType = null);
        ICodeGenerator GeneratorFor<T>() where T : Expr;
    }

    public class CodeGeneratorsFactory : ICodeGeneratorsFactory
    {
        private static CodeGeneratorsFactory _instance;

        private IDictionary<Type, ICodeGenerator> _genartors;

        private CodeGeneratorsFactory() { }


        public ICodeGenerator GeneratorFor(Type type, Type originType = null)
        {
            originType ??= type;


            if (_genartors.TryGetValue(type, out var val))
            {
                return val;
            }
```

```
            return type != typeof(object) ? GeneratorFor(type.BaseType, originType)
                : throw new MissedGeneratorException(originType);
        }

        public ICodeGenerator GeneratorFor<T>() where T : Expr =>
GeneratorFor(typeof(T));

        public static ICodeGeneratorsFactory Instance
        {
            get
            {
                if (_instance != null)
                    return _instance;

                _instance = new CodeGeneratorsFactory();

                _instance._genartors = new Dictionary<Type, ICodeGenerator>
                {
                    { typeof(AssignExpr), new VarDefinitionGenerator(_instance) },
                    { typeof(RecordExpr), new RecordDefinitionGenerator(_instance) },
                    { typeof(FuncInvocationExpr), new FuncCallGenerator(_instance) },
                    { typeof(FuncExpr), new FuncDefinitionGenerator(_instance) },
                    { typeof(RetExpr), new FuncReturnGenerator(_instance) },
                    { typeof(RecordCreationExpr), new
RecordCreationGenerator(_instance) },
                    { typeof(ForExpr), new ForGenerator(_instance) },
                    { typeof(IfElseExpr), new IfElseGenerator(_instance) },
                    { typeof(TernaryIfExpr), new IfElseGenerator(_instance) },
                    { typeof(LambdaExpr), new FuncDefinitionGenerator(_instance) },
                    { typeof(Expr), new SingleExprGenerator(_instance) },
                };

                return _instance;
            }
        }
    }
public static class TypesResolver
    {
        private static Dictionary<string, Type> _types = new Dictionary<string,
Type>
        {
            { "str", typeof(string) },
            { "int", typeof(int) },
            { "bool", typeof(bool) }
        };
```

```
        public static Type Resolve(string name, ModuleBuilder module)
            => _types.TryGetValue(name, out var val) ? val : module.GetType(name);

        public static Type Resolve(TypeExpr expr, ModuleBuilder module) =>
            expr switch
            {


                FuncTypeExpr f => ResolveFuncType(f, module),
                _  => Resolve(expr.Name, module)
            };
        public static Type ResolveFromExpr(Expr expr, CodeGenerationState state,
ICodeGeneratorsFactory factory) =>


expr switch
            {
                VarExpr v => state.ResolveVariableType(v.Name),
                TypedVar t => Resolve(t.Type, state.ModuleBuilder),
                StrExpr _ => typeof(string),
                IntExpr _ => typeof(int),
                BoolExpr _ => typeof(bool),
                AssignExpr a => ResolveFromExpr(a.Value, state, factory),
                RecordCreationExpr r => Resolve(r.Name, state.ModuleBuilder),
                FuncInvocationExpr f => ResolveFuncCall(f, state),
                LambdaExpr l => ResolveLambda(l, state, factory),
                Expr e => TypedLoader.FromValue(e, null, state, factory).Type
            };

        private static Type ResolveFuncCall(FuncInvocationExpr f,
CodeGenerationState state)
        {
            if(!state.DefinedMethods.TryGetValue(f.Name, out var m))
            {
                if (state.TryResolveVariable(f.Name, out var lb))
                {
                    var method = lb.LocalType?.GetMethod("Invoke");
                    return method.ReturnType;
                }
            }

            return m.ReturnType;
        }

        private static Type ResolveLambda(LambdaExpr l, CodeGenerationState
state, ICodeGeneratorsFactory factory)
```

```
        {
            var argsTypes = l.Args.Select(x => Resolve(x.Type, state.ModuleBuilder))
                .Append(ResolveFromExpr(l.Expr, state, factory)).ToArray();

            return typeof(Func<>).MakeGenericType(argsTypes);
        }

        private static Type ResolveFuncType(FuncTypeExpr f, ModuleBuilder
module)
        {
            var argsTypes = f.ArgsTypes.Select(x => Resolve(x, module)).ToArray();
            var returnType = Resolve(f.ReturnType, module);

            return FuncTypeResolver.Resolve(returnType, argsTypes);
        }
    }
public class TypedLoader
    {
        public Type Type { get; }

        public Action Load { get; }

        public TypedLoader(Type type, Action load)
        {


            Type = type;
            Load = load;
        }


        public void Deconstruct(out Type type, out Action load)
        {
            type = Type;
            load = Load;
        }

        public static implicit operator TypedLoader((Type type, Action load) tuple)
            => new TypedLoader(tuple.type, tuple.load);

        public static TypedLoader FromValue(Expr expr, ILGenerator ilGenerator,
CodeGenerationState state, ICodeGeneratorsFactory factory) => expr switch
        {
            VarExpr v => FromMethodScope(v, ilGenerator, state),
            PropExpr p => FromProperty(p, ilGenerator, state, factory),
```

```
        StrExpr str => (typeof(string), (Action)(() =>
ilGenerator?.Emit(OpCodes.Ldstr, str.Value))),
        IntExpr @int => (typeof(int), () => ilGenerator?.Emit(OpCodes.Ldc_I4,
@int.Value)),
        BoolExpr @bool => (typeof(bool), () => ilGenerator?.Emit(@bool.Value ?
OpCodes.Ldc_I4_1 : OpCodes.Ldc_I4_0)),
        RecordCreationExpr record => FromRecordCreation(record, ilGenerator,
state, factory),
        FuncInvocationExpr f => FromFuncCall(f, ilGenerator, state, factory),
        BinaryExpr bin => FromBinaryExpr(bin, ilGenerator, state, factory),
        TernaryIfExpr t => FromTernaryExpr(t, ilGenerator, state, factory),
        LambdaExpr l => FromLambda(l, ilGenerator, state, factory),
        _ => throw new PositionedException(expr.Pos, "Unsupported variable
type")
    };

    public static TypedLoader FromLambda(LambdaExpr lambda, ILGenerator
ilGenerator,
        CodeGenerationState state, ICodeGeneratorsFactory factory)
    {
        factory.GeneratorFor<LambdaExpr>().Generate(lambda, state);
        var method = state.AnonymousMethodsCache.Peek();

        var argsTypes = lambda.Args.Select(x => TypesResolver.Resolve(x.Type,
state.ModuleBuilder)).ToArray();

        var type = FuncTypeResolver.Resolve(method.ReturnType, argsTypes);

        var constructorInfo =
            type.GetConstructor(new [] { typeof(object), typeof(IntPtr) });

        return (type,
            () =>
            {
                ilGenerator.Emit(OpCodes.Ldnull);
                ilGenerator.Emit(OpCodes.Ldftn, method);
                ilGenerator.Emit(OpCodes.Newobj, constructorInfo);
            }
        );
    }

    public static TypedLoader FromTernaryExpr(TernaryIfExpr expr,
ILGenerator ilGenerator,
        CodeGenerationState state, ICodeGeneratorsFactory factory)
    {
```

```
            if (!(expr.Then is ChooseExpr ch))
            {
                throw new Exception("Wrong ternary operator structure");
            }



            var leftType = FromValue(ch.Left, null, state, factory).Type;
            var rightType = FromValue(ch.Left, null, state, factory).Type;

            if (leftType != rightType)
            {
                throw new Exception($"Can not resolve return type of ternary operator.
Left type: {leftType}, right type: {rightType}");
            }

            var generator = factory.GeneratorFor<IfElseExpr>();

            return (leftType, () => generator.Generate(expr, state));
        }

        public static TypedLoader FromBinaryExpr(BinaryExpr expr, ILGenerator
ilGenerator,
            CodeGenerationState state, ICodeGeneratorsFactory factory)
        {
            var leftLoader = FromValue(expr.Left, ilGenerator, state, factory);
            var rightLoader = FromValue(expr.Right, ilGenerator, state, factory);

            (Action<ILGenerator> operation, Type type) = expr switch
            {
                AddExpr _ => leftLoader.Type switch
                {
                    Type t when t == typeof(int) => (OpCodes.Add.EmitSingle(),
typeof(int)),
                    Type t when t == typeof(string) => (EmitStringConcat(),
typeof(string))
                },
                SubtrExpr _ => (OpCodes.Sub.EmitSingle(), typeof(int)),
                MulExpr _ => (OpCodes.Mul.EmitSingle(), typeof(int)),
                DivExpr _ => (OpCodes.Div.EmitSingle(), typeof(int)),
                EqExpr _ => (OpCodes.Ceq.EmitSingle(), typeof(bool)),
                NotEqExpr _ => (EmitNotEq, typeof(bool)),
                LessExpr _ => (OpCodes.Clt.EmitSingle(), typeof(bool)),
                MoreExpr _ => (OpCodes.Cgt.EmitSingle(), typeof(bool)),
```

```
            LessOrEqExpr _ => (OpCodes.Ceq.EmitOr(OpCodes.Clt, leftLoader,
rightLoader), typeof(bool)),
            MoreOrEqExpr _ => (OpCodes.Ceq.EmitOr(OpCodes.Cgt, leftLoader,
rightLoader), typeof(bool)),
            AndExpr _ => (OpCodes.And.EmitSingle(), typeof(bool)),
            OrExpr _ => (OpCodes.Or.EmitSingle(), typeof(bool))
        };

        return LoadWithOperation(leftLoader, rightLoader, type, () =>
operation(ilGenerator));
    }

    private static void EmitNotEq(ILGenerator il)
    {
        il.Emit(OpCodes.Ceq);
        il.Emit(OpCodes.Not);
    }



    public static TypedLoader FromProperty(PropExpr prop, ILGenerator
ilGenerator, CodeGenerationState state, ICodeGeneratorsFactory factory)
    {
        var (varType, varLoader) = FromValue(prop.Expr, ilGenerator, state,
factory);

        varLoader();
        var property = varType.GetProperty(prop.Prop);


        var get = property.GetGetMethod();

        return (property.PropertyType, () => ilGenerator.Emit(OpCodes.Call, get));
    }

    public static TypedLoader FromRecordCreation(RecordCreationExpr expr,
ILGenerator ilGenerator, CodeGenerationState state, ICodeGeneratorsFactory
factory)
    {


        var type = state.ModuleBuilder.GetType(expr.Name);
        var ctor = type.GetConstructors()[0];
        var ctorParams = ctor.GetParameters();

        var providedParams = expr.Props.ToArray();
```

```
            if (ctorParams.Length != providedParams.Length)
                throw new IndexOutOfRangeException("Wrong args");

            for (int i = 0; i < ctorParams.Length; i++)
            {
                FromValue(providedParams[i], ilGenerator, state, factory).Load();
            }

            return (type, () => ilGenerator.Emit(OpCodes.Newobj, ctor));
        }

        public static TypedLoader FromFuncCall(FuncInvocationExpr expr,
ILGenerator ilGenerator, CodeGenerationState state, ICodeGeneratorsFactory
factory)
        {
            var generator = factory.GeneratorFor<FuncInvocationExpr>();

            var fromM = state.DefinedMethods.TryGetValue(expr.Name, out var m);
            var fromV = state.TryResolveVariable(expr.Name, out var lb);

            var returnType = fromM ? m.ReturnType : fromV ?
lb.LocalType?.GetMethod("Invoke")?.ReturnType : throw new Exception("!");
            return (returnType, () => generator.Generate(expr, state));
        }

        public static TypedLoader FromMethodScope(VarExpr expr, ILGenerator
ilGenerator,
            CodeGenerationState state)
        {
            if (state.InnerScope == CodeGenerationScope.Loop)
            {
                if (expr.Name == "index")
                {
                    return (typeof(int), () => ilGenerator?.Emit(OpCodes.Ldloc,
state.DefaultLoopIndex));
                }
                else if(state.LoopIndexes.TryGetValue(expr.Name, out var index))
                {
                    return (typeof(int), () => ilGenerator?.Emit(OpCodes.Ldloc, index));


  }
            }

            if (state.Scope == CodeGenerationScope.Method)
```

```
            {
                if (state.MethodArgs.TryGetValue(expr.Name, out var arg))
                {
                    return (arg.Type, () => arg.EmitLoad?.Invoke(ilGenerator));
                }


                if (state.MethodVariables.TryGetValue(expr.Name, out var v))
                {
                    return (v.LocalType, () => ilGenerator?.Emit(OpCodes.Ldloc, v));
                }


            }
            if (state.MainVariables.TryGetValue(expr.Name, out var mv))
            {
                return (mv.LocalType, () => ilGenerator.Emit(OpCodes.Ldloc, mv));
            }


            throw new Exception("Can not resolve variable");
        }

        private static TypedLoader LoadWithOperation(TypedLoader left,
TypedLoader right, Type returnType, Action EmitOp)
        {
            return new TypedLoader(returnType, () =>
            {
                left.Load();
                right.Load();

                EmitOp();
            }
            );
        }

        private static Action<ILGenerator> EmitStringConcat() => il =>
        {
            var strConcat = typeof(string).GetMethod(nameof(string.Concat),
                new[] { typeof(string), typeof(string) });

            il.EmitCall(OpCodes.Call, strConcat, new[] { typeof(string), typeof(string)
});
        };
    }
public enum SourceType
```

```
    {
        String, File
    }

    public interface ICompiler
    {
        ICompiler WithCodeSource(string source, SourceType sourceType);
        ICompiler WithAssemblyName(string name);

        object Run();



        object Run(out AST ast);

        object RunFromAst(AST ast);
    }

    public class TinyCompiler : ICompiler
    {
        private readonly IASTBuilder _astBuilder;
        private readonly ICodeGeneratorsFactory _codeGeneratorsFactory;

        private string assemblyName, source;


        private SourceType sourceType;

        private TinyCompiler(IASTBuilder astBuilder, ICodeGeneratorsFactory
codeGeneratorsFactory) {
            _astBuilder = astBuilder;
            _codeGeneratorsFactory = codeGeneratorsFactory;

        }

        public static ICompiler Create
        (
            IParserBuilder<Expr> parserBuilder,
            ITokenizer<Expr> tokenizer,
            ICodeGeneratorsFactory codeGeneratorsFactory
        )
        {
            return new TinyCompiler(new ASTBuilder(parserBuilder, tokenizer),
codeGeneratorsFactory);
        }
```

```
        public static ICompiler Create
        (
            IASTBuilder builder,
            ICodeGeneratorsFactory codeGeneratorsFactory
        )
        {
            return new TinyCompiler(builder, codeGeneratorsFactory);
        }


        public object Run()
        {
            return Run(out _);
        }

        public object Run(out AST ast)
        {
            var code = sourceType == SourceType.String ? source :
File.ReadAllText(source);

            ast = _astBuilder.FromStr(code).Build();

            return Run(ast);
        }

        public object RunFromAst(AST ast) => Run(ast);

        public ICompiler WithAssemblyName(string name)



 {

            assemblyName = name;
            return this;
        }

        public ICompiler WithCodeSource(string source, SourceType sourceType)
        {
            this.source = source;
            this.sourceType = sourceType;
            return this;
        }

        private object Run(AST ast)
        {
```

```
        var state = CodeGenerationState.BeginCodeGeneration(assemblyName,
$"{assemblyName}.Module");

        foreach (var expr in ast)
        {
            _codeGeneratorsFactory.GeneratorFor(expr.GetType()).Generate(expr,
state);
        }

        state.MainMethodBuilder.GetILGenerator().Emit(OpCodes.Ret);
        state.ModuleBuilder.CreateGlobalFunctions();

        return state.ModuleBuilder.GetMethod("main").Invoke(null, new
object[0]);
    }
  }
```